Multi Resolution Representations and Interactive Visualization of Huge Unstructured Volume Meshes

Dissertation

der Fakultät für Informations- und Kognitionswissenschaften
der Eberhard-Karls-Universität Tübingen
zur Erlangung des Grades eines
Doktors der Naturwissenschaften
(Dr. rer. nat.)

vorgelegt von
Dipl.-Inform. Ralf Sondershaus
aus Dresden

Tübingen
2007

Tag der mündlichen Qualifikation:	11.07.2007
Dekan:	Prof. Dr. Michael Diehl
1. Berichterstatter:	Prof. Dr. Wolfgang Straßer
2. Berichterstatter:	Prof. Dr. Stefan Gumhold
	(Technische Universität Dresden)

Zusammenfassung

Moderne Simulationen erzeugen immer größer werdende Datensätze und stellen Wissenschaftler vor die Aufgabe, diese Datensätze auszuwerten und zu analysieren. Oft liegen die Daten als reine Zahlenkolonnen vor und erschweren so eine Auswertung. Die Transformation der Daten in eine graphische Darstellung ist ein wichtiges Hilfsmittel, um das Ergebnis einer Simulation oder eine Sammlung gemessener Werte richtig zu interpretieren und auszuwerten. Die vorliegende Arbeit stellt neue Verfahren zur Aufbereitung solch großer Datenstze vor mit dem Ziel, eine interaktive graphische Darstellung zu ermöglichen. Dabei steht insbesondere die Aufbereitung auf Computern mit begrenzten Resourcen im Mittelpunkt der vorgestellten Algorithmen.

Diese Arbeit liefert Beiträge zu drei wichtigen Themen auf dem Gebiet der Aufbereitung von wissenschaftlichen Daten: Reduktion mit garantiertem maximalem Fehler, Bearbeitung großer Datensätze mit begrenzten Ressourcen und interaktive Visualisierung.

Zu Beginn wird ein automatischer Reduzierer vorgestellt, welcher die Anzahl der gespeicherten Elemente verringert und dabei den Approximationsfehler zwischen dem originalen Datensatz und dem reduzierten Datensatz kontrolliert. Der Reduzierer arbeitet global auf der Domäne des Datensatzes, und nicht, wie fast alle bisher bekannten Verfahren, mit vielen kleinen, lokal auf dem Datensatz arbeitenden Schritten. Damit behebt er den Nachteil dieser Verfahren, in lokale Minima zu laufen, und ist bei einem zu den anderen Verfahren vergleichbaren Approximationsfehler ungefähr um den Faktor 3-5 schneller.

Anschließend wird eine Out-of-core Datenstruktur vorgeschlagen, welche ein schnelles Bearbeiten selbst größter Datensätze erlaubt mit einem geringen Bedarf an Hauptspeicher und minimalen Zugriffen auf Sekundärspeicher. Darauf aufbauend werden zwei Datenstrukturen vorgestellt, welche den Datensatz in mehreren Auflösungsstufen speichern. Die erste Struktur speichert die Auflösungen in einer binären Hierarchie während die zweite Struktur das Konzept der Multitriangulation auf Volumennetze anwendet. Eine graphische Darstellung bedient sich dieser zwei Datenstrukturen und erreicht eine interaktive Visualisierung von sehr großen Datensätzen ohne sichtbaren Fehler, auch auf Computern mit begrenzten Resourcen.

Das letzte Kapitel der Arbeit stellt drei Techniken für die Visualisierung vor, welche grundlegende Probleme bei gegenseitigen Verdeckungen adressieren. Für Skalarfelder wird eine punktbasierte Visualisierung vorgeschlagen, welche einen Satz von Isoflächen mit Punkten abtastet und diese Punkte mit Beleuchtung zeichnet. Für Vektorfelder stellt die Arbeit eine Ghosting-Technik

vor, welche einen dichten Satz von Stromlinien semi-transparent darstellt und nur wenige, vom Benutzer wählbare Stromlinien hervorhebt. Im letzten Abschnitt werden Diffusionsflächen eingeführt und ihre Anwendung für die Tensorfeld-Visualisierung mit dem Spezialfall medizinisches MRI diskutiert.

Abstract

Modern scientists must consume ever bigger volumes of data gushing out of supercomputer simulations or high-powered sensors. Often, the data are represented as vast blocks of numbers which need to be transformed into a graphic representation which enables and improves understanding and analyzation. On their way from raw data to interactive visualizion, huge scientific datasets need algorithms out of three areas of research: reduction with a controllable approximation error, out-of-core techniques for huge datasets and interactive visualization of huge datasets with no visual error. All three topics are addressed by this thesis.

At the beginning, an automatic simplification technique is presented which reduces the number of elements of a dataset and controls the approximation error between the original dataset and the reduced dataset. The algorithm consists of a sequence of three steps which work globally on the dataset and improves all known approaches that use a sequence of many small local steps by increasing numerical stability and avoiding runs into local minima. With an achieved approximation error that is comparable to other known approaches, the reducer is about 3-5 times faster.

Next, an out-of-core data structure is introduced which allows for an efficient work on even biggest unstructured datasets with a low consumption of main memory and minimal accesses to secondary storage like hard discs. Based on the out-of-core data structure, two additional data structures are proposed in order to store huge datasets with multiple levels of resolution. The first data structure stores all levels of resolutions in a binary hierarchy whereas the second data structure uses the concept of multi triangulations on unstructured volume datasets.. A rendering framework uses these data structures to achieve an interactive visualization with a visual error controllable by the user. Both structures run on computers with limited ressources even with no visual error.

The last chapter introduces three different visualization techniques addressing fundamental problems of occlusion during rendering. A point-based visualization technique is proposed for scalar fields. A set of isosurfaces is point sampled and the resulting points are rendered with a lighting scheme adapted to the scalar field. A ghosting technique is proposed for vector fields. A dense set of stream lines is calculated but only a few of them are visualized opaque whereas all others are visualized semi-transparent. The user has full control over the selection of the opaque lines. Finally, diffusion surfaces are introduced and it is shown how they can improve the visualization of medical MRI data.

Acknowledgements

This work would not have been possible without the encouraging support of many people. First of all, I want to thank my advisor, Prof. Dr. Dr. E. h. Wolfgang Straßer, for his confidence in my work and for the possibility to work within the Sonderforschungsbereich 382 (SFB 382) of the German Research Council (DFG) which was a great experience. Prof. Straßer has always been able to ask the right questions and to drive research into an appropriate direction.

Special thanks also to Prof. Dr. Stefan Gumhold who suggested to use points as drawing primitives for tensor field visualization and supported the early works on DT-MRI. Furthermore, I encourage everybody to discuss multi resolution data structures and their compact representations with him – Stefan is full of breaking suggestions and offers a complete zoo of great ideas. Discussions with him have heavily influenced chapter 5 of this thesis. His lecture on surfaces influenced the section about topology and space of this thesis.

I want to thank Dr. Uli Bieg from the University of Tübingen for many fruitful disussions about the usefulness of scientific visualizations. In addition, he provided his Sea dataset as an example containing both scalar values as well as vector values. He was the first who used my programs to visualize geological phenomena in his publications.

From EADS, I want to thank Dr. Udo Tremel for giving access to his F16 dataset which is also populated with a lot of scalar and vector quantities. Both the Sea and the F16 dataset have really been used in simulations. They have not only populated my collection of sample datasets by some new datasets never published before but have also influenced the design of my out-of-core algorithms due to their difficult geometrical and topological properties.

I would like to thank all colleagues at GRIS for all their cooperative work especially at the time of lesson preparation. In addition, all of them have took their time to discuss certain ideas and have maintained a working infrastructure including mail servers and network management.

Last but by far not least I want to thank my wife Melanie. She has never given up to support me during the last years, has always come up with fresh new thoughts and has managed to create a lovely home even at hard times.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The purpose of computing is insight not numbers. - Richard W. Hamming

Unstructured tetrahedral meshes have traditionally been used in simulation systems as a fundamental primitive for representing complex volumetric domains. Modeling such domains plays an important role in a wide range of applications, including geophysical simulations and computational fluid dynamics (CFD). Tetrahedral meshes allow a complex domain to be represented at any desired accuracy and are relatively easy to construct given a surface that bounds the domain. As natural extension of triangular meshes to volumes, their accuracy can vary continously over a domain representing very complex regions at an extremely high resolution while other regions are coarsely modeled. Robust and mathematical sound techniques enable tetrahedral meshes to be constructed with all properties desired by simulation software.

With the increasing power and capacity of supercomputers and high-speed storage systems, modern simulation systems can process ever bigger tetrahedral meshes and store their results traditionally as vast files of numbers. If researchers try to extract information out of these numbers directly, they will proceed at a very slow pace or even fail. If the data is transformed into a graphic representation, however, the information can be assimilated at a much faster rate. And if the graphic representation allows the dataset to be explored interactively, information often reveals at an even higher rate. The process of transforming data into a graphic representation is called visualization.

Visualization systems must consume ever bigger data volumes while they often run at commodity hardware like PCs. Although the capacity of PCs has increased steadily over the last years, specialized algorithms and data structures are still required to process and visualize the vast amount of large data volumes. Here, as a very important tool, *automatic model simplification* approximates a highly detailed dataset with a smaller number of data elements. If researchers can accept the loss in fidelity, these tools provide a convenient way to elevate display rates, reduce storage costs and improve provessing speed.

Otherwise, a dataset can be approximated by multiple representations of different complexity. At visualization time, an appropriate *level of detail* is chosen depending on factors such as distance

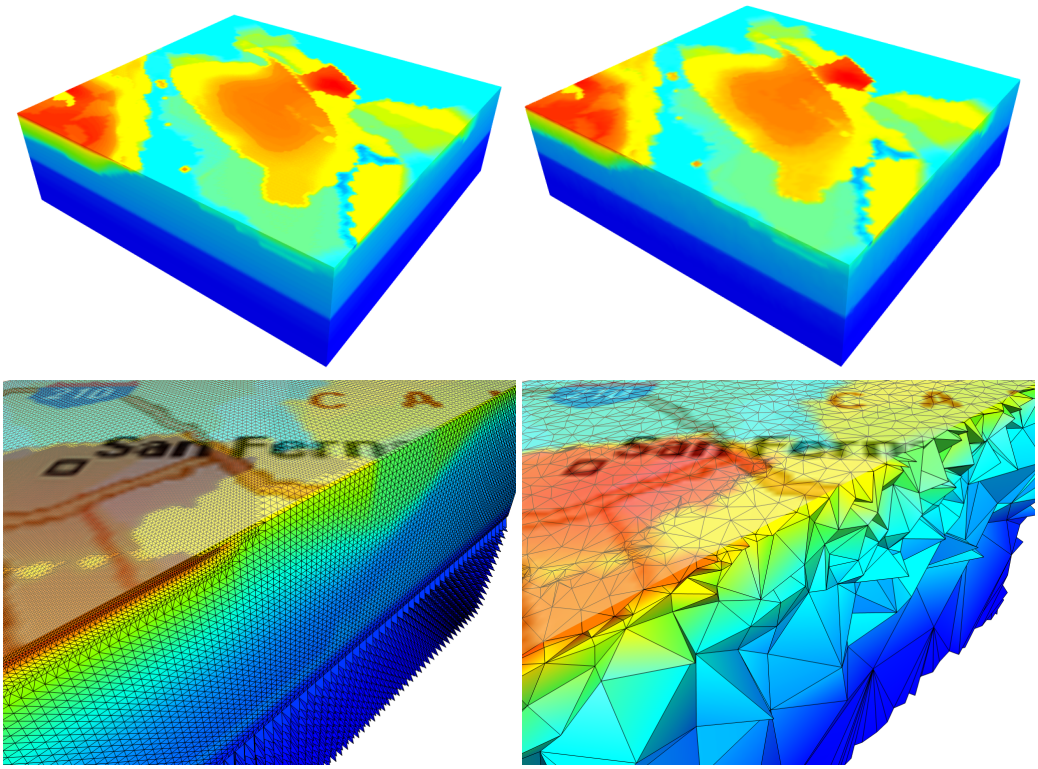


Figure 1.1: Comparision of the fully detailed tetrahedral model (left) and a coarser model (right) produced with the simplification method described in chapter 4.2. The zooms into a particular area of the dataset show its volumetric and non-uniform structure where the simplified model capatures the main properties.

or focus of interest. While distant parts of the dataset require less detail due to foreshortening, parts closer to the viewer need a larger amount of details. But not only visualization systems benefit from multiple levels of detail but also simulation software. Hierarchical approaches like multigrid methods have been shown to decrease computational time by computing a first solution at a coarse level of detail and refining this first solution later on at more detailed levels.

With the growing size of geometric datasets, *out-of-core techniques* become highly important for simplification and level of detail methods. If a dataset is far to big to be stored in main memory (also called in-core memory), such techniques transform it into a representation that allows the dataset to be processed part by part where each part fits into main memory. Since parts must be read from disc and written back to it, a well-designed out-of-core technique minimizes the number of disc accesses while keeping the memory consumption low.

The standard model of *computer-driven visualization* consists of a pipeline with three stages. The first stage retrieves raw data from the simulations or sensors and transforms them into visualization data applying methods of filtering, segmentation, feature extraction, compression or

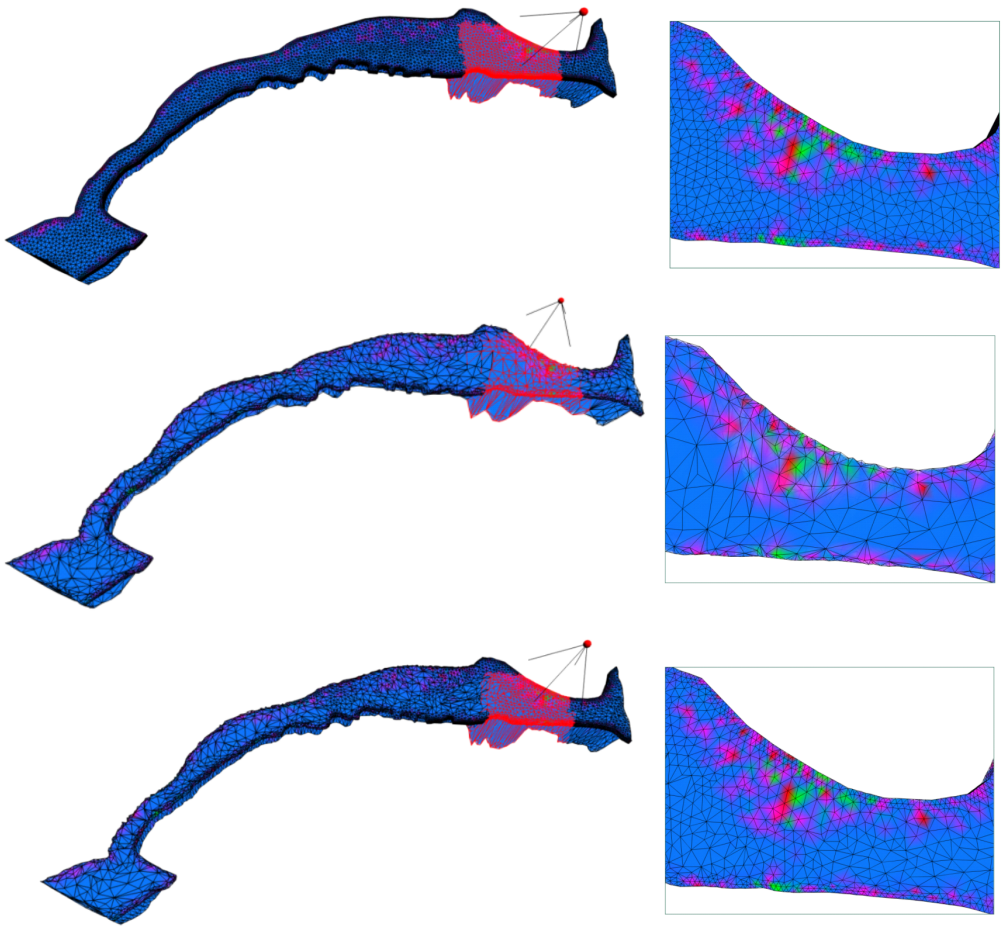


Figure 1.2: A simplified model (middle) may contain a loss in fidelity compared to the original dataset (top) which often reveals in zooms into the dataset. Using multiple representations (bottom) enables a dataset to be visualized at a certain level of detail which can be the finest (produced with the multi-resolution model of chapter 5.2).

simplification. The second stage maps these data into a scene description consisting of graphical primitives like points, lines, triangles, or voxels together with additional parameters like colors or transparancy. The third stage, finally, takes the scene description and generates images or videos based on graphic APIs like OpenGL or DirectX.

For a dimensionality of 1 or 2, traditional plotting software covers most areas of data visualization whereas a tremendous effort has taken place to design useful mappings for data given in 3D space like unstructured tetrahedral meshes. Although the amount of data is huge, ever bigger data volumes show up in simulations or measurements. Not only the sheer size of data volumes is challenging but also the severe occlusion problems that arise from mapping 3D content on a 2D

viewing plane.

Beside tetrahedral meshes, a variety of different types of datasets are used by simulation systems or gush out of sensors. They can be classified by their dimensionality, by the kind of data values that are stored with them including scalars, vectors, or tensors, and, finally, by the structure of how their data values are provided:

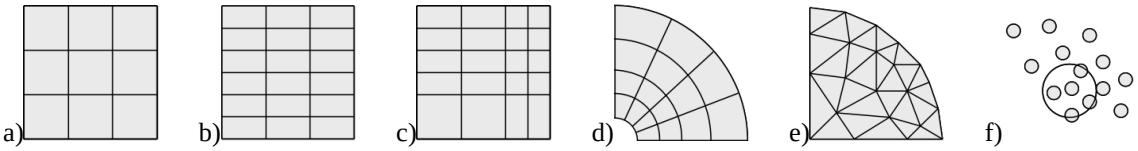


Figure 1.3: Grids from left to right: cartesian, regular, rectilinear, structured, and unstructured. Right most: a point cloud with radius of influence.

a) **Cartesian Grids.** The points are evenly distributed along each axis having the same spacing in all dimensions. The points are highly regular connected by squares in 2D or cubes in 3D. The cubes are also often called *cells* in order to express that the dataset is constructed by many of them. Due to its regularity, a grid cell can be addressed by an index (i, j, k) and the points have coordinates $(i \times d, j \times d, k \times d)$ for a fixed distance d.

b) **Regular Grids.** Basically similar to cartesian grids, regular grids can have different spacings for each axis, that is, each axis has its own sampling distance. The data set is composed of cuboids (in contrast to cubes in cartesian grids). Famous examples for regular grids are medical datasets resulting from CT or MRI scans. Typically, these modalities measure 2D slices at a very high resolution and construct the data set as a stack of such 2D slices where the distance between two slices is lower than the resolution of a single slice. Again, a grid cell can be addressed by an index (i, j, k) and the points have coordinates $(i \times d_x, j \times d_y, k \times d_z)$ for fixed distances d_x, d_y and d_z.

c) **Rectilinear Grids.** As the spacing is constant for each dimension in cartesian and regular grids, it varies in rectilinear grids. The data sets are composed of cuboids with varying sizes. Although the cells may still be indexed by integers as in regular and cartesian grids, the mapping from indices to point coordinates is not regular anymore. Graphs with logarithmically scaled axes are typical examples.

d) **Structured Grids.** The grid points are connected like in regular grids but the locations can be chosen arbitrarily deforming the rectangular cuboidal cells to 2D quadriliterals or 3D hexahedral cells. The mapping between a point index and its coordinates is free. Many simulations also benefit from structured grids with curved connections between sample locations resulting in even more deformed curved cuboidal cells. Such curvilinear grids are very popular as finite elements meshes. But although the grid cells can adapt to a given

boundary very well, the resolution of the grid cannot be changed easily. Thus, most often hybrid grids are used combining the advantages of both structured and unstructures grids.

e) **Unstructured Grids.** Both the sample locations and their connections are arbitrary. The cell types can differ within the grid. Tetrahedra and prisms are frequently used cell types. For rendering purposes, an unstructured grid is often transformed into a purely tetrahedral mesh. Unstructured grids can adapt to given boundaries very well and can change the resolution freely due to their unstructured nature.

f) **Point Clouds.** Finally, the sample locations can be arbitrary without a fixed connection. For instance, Smoothed Particle Hydrodynamics (SPH) simulations use a cloud of sample points. The sample points can move freely and interact with other near-by sample points. Therefore, each sample point has to find all other sample points that are within a region of influence. Similar, range scanning systems generate unstructured point clouds which need to be interpreted as a surface later on.

1.1 Problem Statement

On their way from raw data to interactive visualizion, huge scientific datasets need algorithms out of several areas of research. First, automatic simplification techniques reduce the complexity of datasets with the goal to compute an approximation with a minimal approximation error. The error between two datasets is measured with a global error metric whose choice depends strongly on the application type. For instance, polygonal datasets have to use other metrics than volume datasets and even polygonal datasets require different metrics depending on subsequent processing stages like visualization or compression. Because global error metrics are often computational expensive and the calculation of a minimal approximation has been shown to be NP-hard [AS98], local error metrics are often used measuring local errors only. The design of metrics that are computational easy to evaluate and still yield good global errors is still an open research area. Chapter 4 discusses automatic simplifiers and introduces a new rapid simplifier for huge tetrahedral meshes together with an error metric that can be computed quickly.

As the increasing size of scientific datasets outpaces the progress in processor speed and available memory, out-of-core techniques have become an important area of research not only to enable huge datasets to be processed at all but also to improve the processing speed due to a better utilization of computational resources. I will describe an out-of-core technique that is computational easy to create and resource-friendly.

In addition to pure simplifications, the design of data structures and algorithms for creating multiple levels of detail is highly important for interactive visualization of datasets *without* loss of fidelity. The design goal for such data structures comprises both compactness and rapid adaptation of the dataset to changing parameters while the dataset remains a valid approximation. Due to changes of computer architectures and the increasing gap between processing speed and memory

speed, many older algorithms need to be improved to reflect these changes or need to be replaced by new algorithms and data structures.

Finally, visualization techniques need to address the severe occlusion problems that arise with the mapping of 3D content onto a 2D viewing plane and establish a field of research of its own.

1.2 Contributions

This thesis describes new algorithms for rapid simplification of unstructured tetrahedral meshes providing small approximation errors, a total of four different multi-resolution models based on a new out-of-core data structure, and, finally, visualization techniques for scalar fields, vector fields and MRI tensor fields.

1. *Simplification, chapter 4.* As a prerequisite for multi-resolution models, some issues of simplifiers for tetrahedral meshes are discussed in § 4.1. An algorithm based on iterative edge collapses is presented that does not need to maintain all edges within a single large queue but does need to maintain some edges only lying on an advancing front. In this manner, the simplifier need less memory and still simplifies a mesh uniformly. It is therefore perfectly suited for the construction of multi-resolution models.
   § 4.2 presents a novel approach for the rapid simplification of large tetrahedral meshes which generates approximations with simple point sampling and an independent boundary simplification. The produced global attribute error of the volume mesh is similar to other state-of-the are approaches based on iterative edge collapses with quadrics but performs as twice as fast [SS06a]. In addition, the boundary triangular mesh is simplified without introducing self-intersections. Currently, this simplifier is the fastest known simplification approach.

2. *Multi-Resolution Models I, chapter 5.* Based on the very fine-grained coarsening operation edge collapse and its inverse refinement operation, vertex split, two new multi-resolution models are introduced. The first model, § 5.2, exploits the redundancy in the connectivity information of a mesh in order to split a vertex. As main contribution, all tetrahedra keep their vertex indices stored during edge collapses and use these indices later in order to perform vertex splits correctly by a fast inspection of these vertex indices. Thereby, new tetrahedra can be simply created by predicting their indices from surrounding tetrahedra [SS05b]. In addition, the index prediction allows the update information to be compressed very well.
   The second model, § 5.3, is restricted to half-edge collapses and splits resulting in a very compact multi-resolution model [SS05a]. As an extension of FastMesh [Paj01], it maintains a half-edge data structure where no element is delete physically during an edge collapse but remains stored. In this manner, the data structure allows vertex splits and collapses to be computed fast and robust with highly efficient run-time checks for their validity.

3. *Out-of-core Data Structure, chapter 6.* § 6.1 presents a data structure that enables the out-of-core processing of huge meshes by processing a sequence of compact parts of a mesh, so-called segments. It is evaluated by out-of-core simplification and out-of-core construction of multi-resolution models [SS06c]. Its beauty is based on an easy process of creation and its streaming capailities that not only allow for an out-of-core processing of large datasets but also for an improved processing speed.

4. *Multi-Resolution Models II, chapter 6.* Two multi-resolution models are presented which work by replacing segments of one resolution by segments of another resolution and are based on the out-of-core data structure described above.
   The first model in § 6.2 exploits a binary hierarchy of segments. It is advantageous for finite element simulations and is easy to implement [SS06c]. It is the first published data structure for huge unstructured tetrahedral meshes being tailored for segment-based multi-resolution models.
   The second model in § 6.3 applies the concepts of multi-triangulations to huge tetrahedral meshes. Here, the concept of rotating octrees enables a segmentation of the mesh which allows for a very smooth simplification of the mesh and an *automatic* transformation into a multi-resolution model [SS06b]. Additionally, the segments are stored compressed and can be decompressed on the fly using a modified cut-border machine.

5. *Visualization Techniques, chapter 7.* First, § 7.2 introduces a rendering modality for scalar fields which basically computes a set of isosurfaces and point samples them. Thereby, points on a single isosurface are closer to each other than points on different isosurfaces. Only points are rendered. The rendering of none-dense point clouds enables deep looks into inner structures of the dataset without any depth-ordering of mesh elements [SS05b].
   Second, a ghosting technique is introduced in § 7.3 for vector field visualization in order to decrease occlusion and to increase the benefit of interactive exploration. Out of a dense set of stream lines, the user selects a bundle of stream lines that are visualized opaquely while all other stream lines are rendered semi-transparent and can be recognized as ghosts. An interactive exploration of important regions of a vector field becomes possible.
   The stream lines are visualized with different halo techniques. As a main contribution, the halos carry additional information which allows for an easy visualization of the flow direction [SS06b].
   Finally, § 7.4 introduces the concept of diffusion surfaces [SG03] for tensor fields with a strong application in MRI datasets and presents a point-based visualization technique for these similar to § 7.2.

Chapter 2

Basics on Spaces

Space is not a passive vacuum, but has properties that impose powerful constraints on any structure that inhabits it. - Arthur Loeb

In order to establish a unique way to explain ideas in subsequent chapters, some of the most common notations and concepts are introduced in this chapter. Starting with topology as a fundamental mathematical concept to describe spaces like surfaces or volumes, simplicial complexes and mesh concepts are presented together with their frequently used nomenclature. For a more detailed introduction we refer the interested reader to [Mn98, Ede01].

2.1 Topology

The concept of topological spaces is an important tool in order to capture structural features of geometric spaces and to formulate terms like similarity of surfaces or volumes in a mathematical solid way. It is based on point sets and how the points are related to each other.

A *Topological Space* $\underline{\mathbf{X}}$ is a pair $(X, \mathbf{X})$ comprising a point set X and a set $\mathbf{X}$ of open subsets $A \subseteq X$ such that

1. $\mathbf{X}$ contains both the empty set and the point set X:
   $\oslash, X \in \mathbf{X}$
2. If a collection of open subsets is in $\mathbf{X}$, so their union:
   $\mathbf{Z} \subseteq \mathbf{X} \Rightarrow \bigcup \mathbf{Z} \in \mathbf{X}$
3. If a finite collection of open subsets is in $\mathbf{X}$, so their cut:
   $\mathbf{Z} \subseteq \mathbf{X}$, $\mathbf{Z}$ finite $\Rightarrow \bigcap \mathbf{Z} \in \mathbf{X}$

The open subsets can be understood as a way to describe neighborhoods of points and thus the structural properties of space. For instance, the unit circle $C = \left\{\mathbf{p} \in \mathbf{R}^2 \mid \|\mathbf{p}\| = 1\right\}$ is a topological space with the point set $X = C$ and the set of open subsets comprising all open circle arcs.

As another example, the open unit sphere $S = \{\mathbf{p} \in \mathbf{R}^3 \mid \|\mathbf{p}\| < 1\}$ forms a topological space with the point set $X = S$ and the set of open spheres centered at 0 as set of subsets. Finally, the well-known $\mathbf{R}^d$ is a topological space which is made up of all its points $\mathbf{p} \in \mathbf{R}^d = X$ and open balls $B_r = \{\mathbf{p} \in \mathbf{R}^d \mid \|\mathbf{p}\| < r\}$ that form the set $\mathbf{X} = \{B_r \subset \mathbf{R}^d \mid r \in \mathbf{R}\}$.

A *Topological Subspace* of a topological space $(X, \mathbf{X})$ is a subset of the points $Y \subseteq X$ with a subset of the set collection $\mathbf{Y} \subseteq \mathbf{X}$ such that

1. $\mathbf{Y} = \{Y \bigcap A \mid A \in \mathbf{X}\}$

Topological subspaces allow spaces to be constructed that can contain their boundaries. For instance, the closed unit sphere is a topological subspace of $\mathbf{R}^3$ with $Y = S = \{\mathbf{p} \in \mathbf{R}^3 \mid \|\mathbf{p}\| \leq 1\}$. Informally, the subspace inherits the structural properties from the topological space, i.e. the topology.

Two topological spaces can be compared by looking at the kind of function that transforms them into each other. Good-natured functions preserve the neighborhood of a point and are called continuous. Formally, a function f between two topological spaces $\underline{\mathbf{X}}$ and $\underline{\mathbf{Y}}$ is *continuous* iff the preimage of any open subset is an open subset. A continuous function is also referred to as a *mapping*. An open subset is always mapped onto an open subset (and vice versa). If such a subset is considered as a neighborhood of any of its points, a continuous function preserves this neighboorhood and does not break it apart.

In order to describe complete mappings between two topological spaces, the term of a homeomorphic function extends a continuous function by forcing bijectivity, continuouity and the existance of an inverse mapping. Spaces that can be transformed into each other by homeomorphic functions are topological equivalent, that is, they are connected the same way. Formally, a function f between two topological spaces $\underline{\mathbf{X}}$ and $\underline{\mathbf{Y}}$ is called *homeomorphic*, iff f is bijective and continuous and its inverse f^{-1} exists and is continuous. A homeomorphic function f is called a *homeomorphism*.

Two topological spaces $\underline{\mathbf{X}}$ and $\underline{\mathbf{Y}}$ are called *homeomorph* or *topological equivalent*, iff there exists a homeomorphism between $\underline{\mathbf{X}}$ and $\underline{\mathbf{Y}}$.

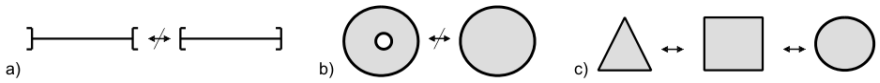


Figure 2.1: (a) The left interval contains its extreme values while the right interval doesn't. The two intervals are not homeomorph. (b) A donut and a circle are not homeomorph. (c) The triangle, quadrilateral and circle are homeomorph.

The property of manifoldness describes topological spaces having nice structural properties. A topological space $\underline{\mathbf{X}} = (X, \mathbf{X})$ is called *k-manifold*, iff there exists for any point $x \in X$ an open set $U \in \mathbf{X}$ which includes x and is homeomorph to $\mathbf{R}^k$.

In order to describe manifold topological spaces with boundaries, the term manifoldness can be extended to consider the half-space $H^k = \{\mathbf{x} = (x_1, ..., x_k) \in \mathbf{R}^k \mid x_j > 0\}$. A topological

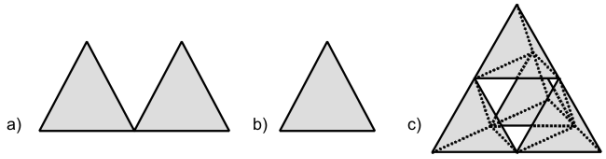


Figure 2.2: (a) Two triangles connect at a non-manifold point. (b) A single triangle is manifold (with boundary). (c) Four tetrahedra connect at six non-manifold vertices.

space $\underline{\mathbf{X}}$ is k*-manifold with border*, iff there exists for any point $x \in X$ a open set $U \in \mathbf{X}$ which includes x and is homeomorph to $\mathbf{R}^k$ or H^k. For instance, the circle in figure 2.1 is 2-manifold with border.

2.2 Simplicial Complexes

While topological spaces are defined on point clouds, the concept of simplicial complexes gives a *combinatorial* structure to these spaces and can be understood as a way to construct topological spaces. Simplicial complexes are based on affine combinations of points.

An *affine combination* of points $\mathbf{p}_1, ..., \mathbf{p}_{k+1}$ is a linear combination $\sum_{i=1}^{k+1} \alpha_i \mathbf{p}_i$ with $0 \leq \alpha_i \leq 1$ and $\sum_{i=1}^{k+1} \alpha_i = 1$. A set of $k+1$ points is *affine independent* iff no affine space of dimension i contains more than $i+1$ of the points, and this is true for all i.

A k*-simplex* comprises the affine combinations of $k+1$ affine independent points. Thereby, k is the *dimension* of the k-simplex. A point as a 0-simplex, a line is a 1-simplex, a triangle is a 2-simplex, a tetrahedron is a 3-simplex, and so on as visualized in figure 2.3.

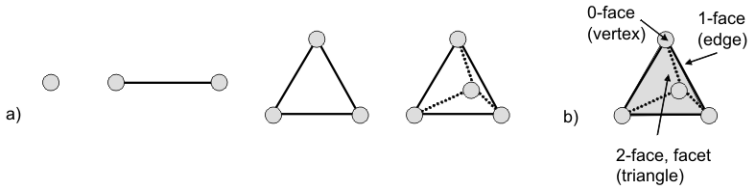


Figure 2.3: (a) From left to right: point, line, triangle, and tetrahedron. (b) A tetrahedron consists of several faces.

A simplex can contain other simplices of smaller dimension. Formally, given a simplex σ of dimension k and any $l \leq k$, the affine combination of $l+1$ nodes span another l-simplex τ. τ is called a l*-face* of σ denoted as $\tau \leq \sigma$. A face of dimension $l = k-1$ is called a *boundary face* or *facet*. A simplex that is not a face of another simplex is called a *maximal simplex*.

A simplicial complex is now a collection of simplices that connect in a particular way, namely, a *Simplicial Complex* is a set K of simplices with

1. Every face of a simplex in K is in K:

$\sigma \in K \wedge \tau \leq \sigma \Rightarrow \tau \in K$

2. The intersection of any two simplices of K is a face of each of them:
   $\sigma, \tau \in K \Rightarrow \sigma \cap \tau \leq \sigma, \tau$

The dimension of a simplicial complex K is the maximum of the dimensions of all its simplices. Every d-simplex of a simplicial complex with dimension d is called a *cell*. I will generally assume that K is a *pure complex*, that is, no l-simplex occurs is K except as a l-face of a d-simplex.

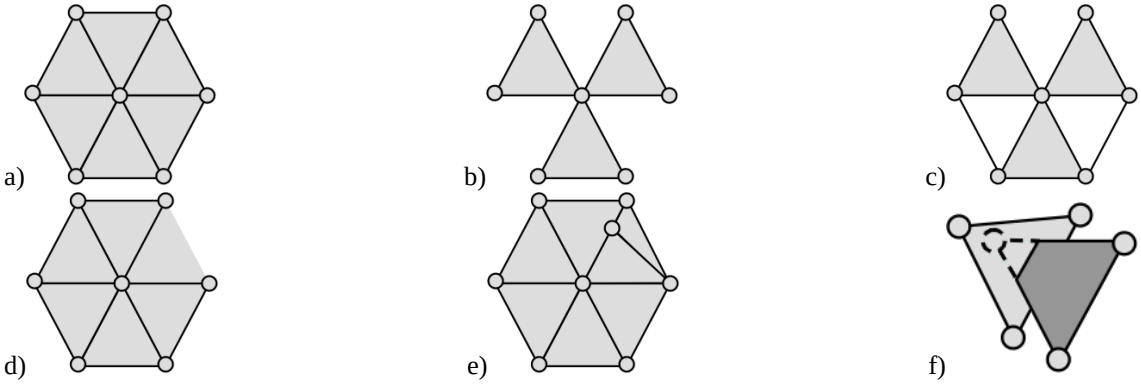


Figure 2.4: (a-c) are simplicial complexes where (a) is manifold (with boundary), (b) is non-manifold at its center vertex, and (c) is not a maximal complex due to the edges that are maximal simplices. (d-f) are not simplicial complexes where (d) misses an edge and violates condition (1), (e) and (f) violate condition (2).

Let us stress that the above definition declares a simplicial complex K to be combinatorial in nature, namely a *set* of simplices satisfying certain conditions, but not to be a subset of $\mathbf{R}^d$. This is done by its *geometric realization* $|K|$ which defines the underlying space as the union of its simplices together with the subspace topology of $\mathbf{R}^d$:

$$|K| = \left\{ \mathbf{x} \in \mathbf{R}^d | \mathbf{x} \in \sigma \in K \right\}$$

The field of algebraic topology extends this combinatorial concept by defining abstract simplicial complexes. For a given set $V = \{1, 2, 3, ...\}$ of vertex numbers, an *Abstract Simplicial Complex* is a finite nonempty family A of subsets of V with

- Every subset C of a set B in A is itself contained in A:
  $B \in A \wedge C \subseteq B \Rightarrow C \in A$

The subsets of size 1 correspond to the vertices, i.e. $\{1\}, \{2\}, \{3\}, ...$, the subsets of size 2 correspond to edges, i.e. $\{1, 2\}, \{1, 4\}, \{3, 5\}, ...$, the subsets of size 3 correspond to triangles and the subsets of size 4 correspond to tetrahedra.

The definition of an abstract simplicial complex describes how simplices of varying dimensions can be connected to each other. A geometric realization is then defined by a mapping from the vertex set V onto points in R^d which can be continued to simplices of higher dimensions.

2.3 Meshes

Meshes also give combinatorial structure to a space by decomposing the space into mesh elements like vertices, edges, triangles or tetrahedra, and their embedding into the space. Formally, a mesh can be separated into its connectivity which describes neighborhoods – the incidencies – between mesh elements, and its geometry which specifies the positions of mesh elements in $\mathbf{R}^d$.

A mesh whose connectivity fulfills the conditions of an abstract simplicial complex is called an abstract simplicial mesh while a mesh whose connectivity fulfills the conditions of a simplicial complex is called a simplicial mesh. All other meshes (like hexahedral grids) are said to be non-simplicial meshes. Abstract simplicial meshes are defined purely on their connectivity information without considering the geometry while simplicial meshes consider both connectivity and geometry. As example, tetrahedral meshes with self-intersections can be abstract simplicial meshes while tetrahedral meshes without self-intersections are simplicial meshes.

The incidency relation describes how mesh elements of different types are connected to each other. For (abstract) simplicial meshes, the incidency relation is defined as

1. An edge $\{p, q\}$ is incident to both its extreme vertices $\{p\}$ and $\{q\}$ (and vice versa).
2. A triangle $\{p, q, r\}$ is incident to
   - three edges $\{p, q\}$, $\{q, r\}$ and $\{r, p\}$ (and vice versa).
   - three vertices $\{p\}$, $\{q\}$ and $\{r\}$ (and vice versa).
3. A tetrahedron $\{p, q, r, s\}$ is incident to
   - four triangles $\{p, q, r\}$, $\{p, q, s\}$, $\{p, r, s\}$ and $\{q, r, s\}$ (and vise versa).
   - six edges $\{p, q\}$, $\{p, r\}$, $\{p, s\}$, $\{q, r\}$, $\{q, s\}$ and $\{r, s\}$ (and vise versa).
   - four vertices $\{p\}$, $\{q\}$, $\{r\}$ and $\{s\}$ (and vise versa).

The term *adjacency* describes how mesh elements of the same type are related to each other

1. Two vertices $\{p\}$ and $\{q\}$ are adjacent iff there exists an edge $\{p, q\}$.
2. Two edges are adjacent iff there exists a vertex incident to both.
3. Two triangles are adjacent iff there exists an edge incident to both of them.

The mesh elements are embedded into space by their geometry which maps each mesh element to its positions in space:

1. A vertex is mapped to a point in $\mathbf{R}^3$.
2. An edge is mapped to an arc connecting its both extreme vertices. In the simplest case, the edge is a straight line segment.

3. A triangle is mapped to the space of the (possibly bended) triangle enclosed by its incident edges.

4. A tetrahedron is mapped the space of the (possibly bended) tetrahedron enclosed by its incident triangles.

The *orientation* of a triangle or tetrahedron establishs an order for its indices, that is, the (unordered) set $\{p, q, r\}$ of a triangle becomes the ordered triple (p, q, r) (and similar quadruples for tetrahedra). Permutations of the same parity refer to the same triangle (or tetrahedron), e.g. (p, q, r) equals (q, r, p) but (p, q, r) and (q, p, r) are triangles of opposite orientation.

Often, the orientation of a tetrahedron is chosen such that its volume becomes positive and the normals of the tetrahedral faces point outwards.

A manifold triangle mesh is said to be *orientable* if there exists a configuration of orientations for its triangles such that for each edge $\{p, q\}$ there exists no pair of triangles with (p, q, r) and (p, q, s). This means, if two triangles are adjacent to each other over an edge $\{p, q\}$, one triangle must be incident to the oriented edge (p, q) while the adjacent triangle must be incident to the reverse oriented edge (q, p). The Möbius strip is a famous example of a manifold surface that is not orientable. Non-self intersecting triangle meshes in a plane are always orientable which is also true for non-self intersecting tetrahedral meshes in $\mathbf{R}^3$.

The neighborhood of mesh elements (or simplices) in abstract simplicial meshes can be enumerated using the following definitions. The *closure* of a set S of simplices is the smallest possible subcomplex C containing all elements of S:

$$C = Cl(S) = \{\tau \in K | \tau \leq \sigma \in S\}$$

The *star* of a simplex τ is the collection of all simplices that have τ as a face:

$$St(\tau) = \{\sigma \in K | \tau \leq \sigma\}$$

The *link* of a simplex τ is the outer region of its star:

$$Lk(\tau) = \{\sigma \in Cl(St(\tau)) | \sigma \cap \tau = \oslash\}$$

The term manifoldness for (abstract) simplicial mesehs can be directly retrieved from topology. For this, the mesh is *realized* in $\mathbf{R}^3$ by defining its underlying space as the union of its geometric embeddings together with the subspace topology of R^d, similar to simplicial complexes. A mesh is called *manifold* if it is manifold in topological sense. Note that the meshing community often weakens this topological definition of manifoldness by using a definition based on the connectivity alone leading to the term of potential manifoldness as follows.

A triangular mesh is said to be *potential manifold* iff each vertex is incident to a non empty set of triangles that from a closed cycle and each edge is incident to exactly two triangles (or, with boundaries, to one or two triangles).

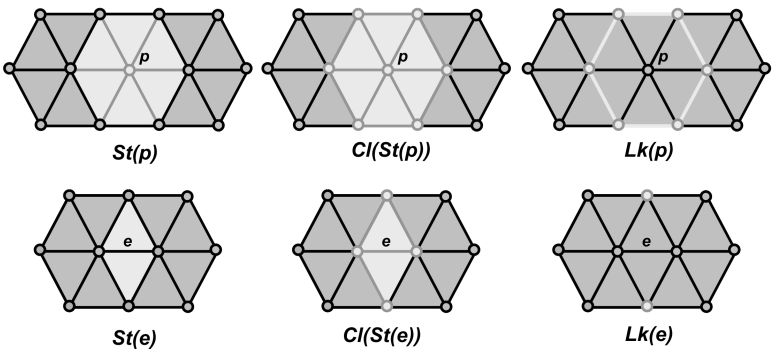


Figure 2.5: The light-gray mesh elements belong to the closure, star and link for a vertex p and an edge e.

A tetrahedral mesh is said to be potential manifold iff the link of each vertex forms a potential manifold triangle mesh, each edges is incident to a non empty set of tetrahedra that from a closed cycle and each face is incident to exactly two tetrahedra (or, with boundaries, to one or two tetrahedra).

Tetrahedral meshes normally have a *boundary*. We define a face $\{p, q, r\}$ to be a *boundary face* if it is incident to exactly one tetrahedron $\{p, q, r, s\}$. Similar, an edge $\{p, q\}$ is a *boundary edge* if it is incident to a boundary face $\{p, q, r\}$, and a vertex $\{p\}$ is a *boundary vertex* if it is incident to a boundary edge $\{p, q\}$.

Two meshes are called *conformal* iff the underlying topological spaces are homeomorph.

2.3.1 Tetrahedral Meshes for Simulations

All meshes covered by algorithms of this thesis are simplicial tetrahedral meshes. They discretize a *domain* Ω which does not need to be convex. In most cases, Ω will be highly non–convex, see § 2.4. The meshes are typically used by simulations and do not only disretize a volumetric domain but also an *attribute field*. Formally, the mesh enables a n-valued function to be approximated by a linear function Φ. Therefore, each vertex stores an n-dimensional attribute value with $n \geq 1$. The attribute values can be scalars or vectors. They are linearily interpolated within a tetrahedron in order to define a linear continuous approximating function Φ. Note that linear interpolation inside a tetrahedron is a frequently used interpolation scheme but other interpolation schemes exist.

2.3.2 Nomenclature

I will use the following nomenclature to describe mesh elements and their geometric embeddings:

- $\{p\}$ or simply p is a single vertex index,
- $\mathbf{p}$ is the geometric realization of a point p (or a point in $\mathbf{R}^d$ in general),

- $e = \{p, q\}$ is a single edge comprising its extreme vertices p and q,
- $e = (p, q) \neq (q, p)$ is a single oriented edge starting from p and heading to q,
- $\{p, q, r\}$ is a single triangle,
- (p, q, r) is an oriented single triangle,
- $\{p, q, r, s\}$ is a single tetrahedron,
- (p, q, r, s) is an oriented single tetrahedron,
- $Cl(S)$ is the closure of a set S of mesh elements,
- $St(p), St(e)$ is the star of a vertex p or an edge e,
- $Lk(p), Lk(e)$ is the link of a vertex p or an edge e,
- Ω is a domain of a tetrahedral mesh,
- Φ is the attribute field of a tetrahedral mesh,
- $\mathbf{a} = \Phi(\mathbf{p})$ are all attribute values given at point $\mathbf{p}$,
- b (non-bold) is a scalar,
- $\mathbf{b}$ (bold) is a point in $\mathbf{R}^d$,
- $\mathbf{A}$ (bold) is a matrix.

2.4 Evaluation Datasets

The collection of datasets that have been used to evaluate my methods are listed in table 2.1. The Sea dataset models an ancient ocean that covered the Alpine foredeep between southeastern France and eastern Austria including southern Germany and parts of Switzerland in the so-called Ottnangian earth time about some million years ago. The scalar values describe two turbulence variables, temperature and salinity. The vector value contains the velocity of water currents within the water. The currents have transported sediments that settled down to the ground forming molasse regions which are part of today's north alpine foreland basin.

The Post dataset was used to simulate how air flows around an upright post that is fixed in the earth. The F16 models a F16-like fighter aircraft that is exposed to air. Its two scalar values contain air density and total energy, the vector value describes the flow direction of air. The simulation has been carried out on parallel computation nodes. The dataset is courtesy of Udo Tremel, EADS-Military.

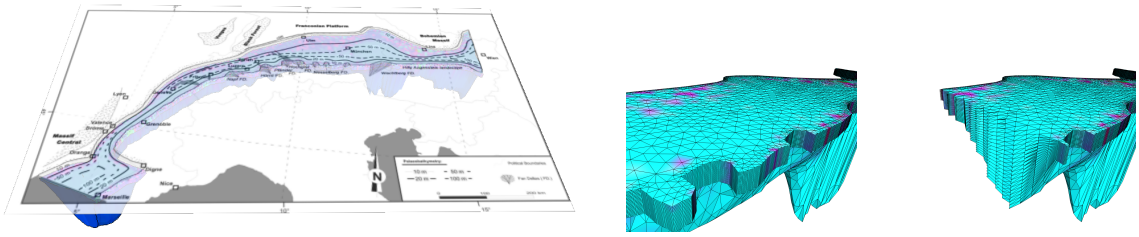


Figure 2.6: The Sea dataset models an ancient ocean that covered the Alpine foredeep between southeastern France and eastern Austria.

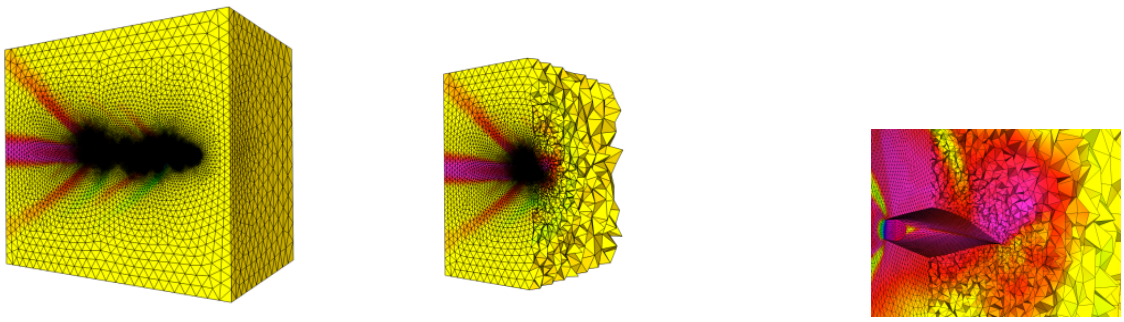


Figure 2.7: The Fighter dataset models an aircraft in a wind tunnel.

The Rbl dataset is a portion of an endoplasmic reticulum in a cell and courtesy of Alex Smith and Bridget Wilson from University of Mexico and Jason Shepherd and Shawn Means of Sandia National Laboratory.

The Fighter dataset comes from a wind tunnel model of a fighter aircraft and is courtesy of Neely and Batina from NASA.

The Earthquake mesh discretizes the San Fernando valley for simulations of earthquakes which were carried out as part of the Quake project at Carnegie Mellon, USA. The simulation run on a Cray T3D at the Pittsburgh Supercomputing Center. The three scalar values represent p-wave velocity (km/h), s-wave velocity (km/h) and density (g/cm^3). The model corresponds to a volume of earth roughly 50 km $\times$ 50 km $\times$ 10 km. The file size of this dataset is 104 MB for geometry and 974 MB for connectivity and attributes, so the overall size of the file is roughly 1 GB. For a detailed description of earthquake theory and different wave types the interested reader is referred to [Bie06].

Tables 2.1 and 2.2 report the combinatorial properties of all these datasets while table 2.3 reports both combinatorial and geometrical properties. Note the highly unbalanced geometric properties of especially the F16 dataset.

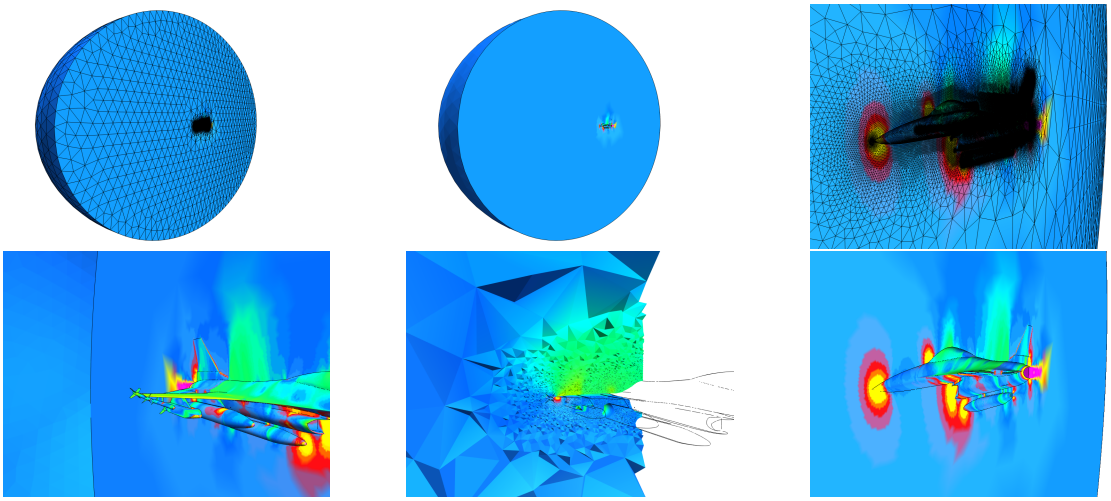


Figure 2.8: A full view of the F16 dataset, with and without edges. A zoom into the region that models the aircraft itself, again with and without edges. A view from inside the dataset onto its boundary reveals all the full details. Small tetrahedra model the air surrounding the aircraft.

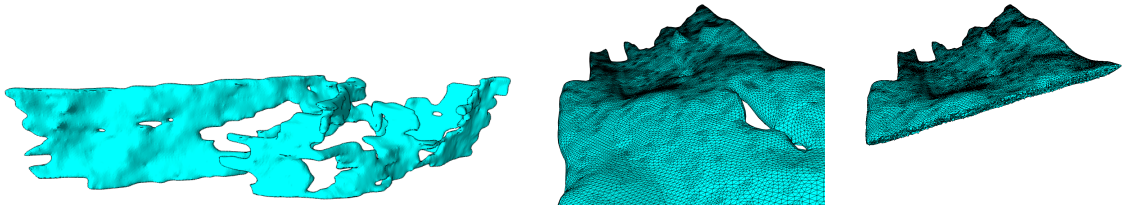


Figure 2.9: The Rbl dataset.

Name	# Vertices	# Tetra	# Edges	# Faces	# Border Triangles	# Vertex Attributes	
						Scalar	Vector
Sea	102,165	524,640	655,228	1,077,704	56,848	4	1
Post	108,300	624,153	740,850	1,256,703	16,796	1	-
Neghip	262,144	1,250,235	1,536,192	2,524,283	47,628	1	-
Fighter	256,614	1,403,504	1,701,869	2,848,760	83,504	1	-
Rbl	730,273	3,886,728	4,779,497	7,935,936	324,960	1	-
F16	1,124,648	6,345,709	7,625,318	12,846,379	309,932	2	1
Earthquake	2,461,694	13,980,162	16,684,112	28,202,580	484,514	3	-

Table 2.1: These datasets were used to evaluate algorithms.

Name	v:	t:	e:	f
Sea	1	5.13	6.41	10.55
Post	1	5.76	6.84	11.60
Neghip	1	4.77	5.86	9.63
Fighter	1	5.47	6.63	11.10
Rbl	1	5.32	6.54	10.87
F16	1	5.64	6.78	11.42
Earthquake	1	5.68	6.78	11.46

Table 2.2: The relations of vertices to tetrahedra to edges to faces, v:t:e:f.

Name	Per Vertex						Edge Length		
	# Incident Tets			# Adjacent Vertices					
	min	max	avg	min	max	avg	min	max	$\frac{\min}{\max}$
Sea	1	36	20.50	3	20	12.83	441.48	118263.000	267.88
Post	3	48	23.05	5	26	13.68	0.00084	3.190	3,798.50
Neghip	1	32	19.07	3	18	11.72	0.031	0.045	1.41
Fighter	1	44	21.87	3	24	13.26	0.139	192.530	1,384.16
Rbl	2	46	21.29	4	25	13.10	0.023	2.190	93.85
F16	1	90	22.57	3	47	13.56	0.00038	12.060	32,151.50
Earthquake	1	64	22.70	3	34	13.56	0.042	0.608	14.38

Table 2.3: The vertex valences and geometric properties.

Chapter 3

Previous Work

It is not knowledge, but the act of learning, not possession but the act of getting there, which grants the greatest enjoyment. - Carl Friedrich Gauß

This chapter is a survey of algorithms for mesh simplification as well as multi resolution techniques and introduces the fundamental previous works of both areas as a basis for the algorithms covered by this thesis. The chapter is divided into two parts where the first part covers simplification algorithms while the second part covers multi resolution techniques. The following chapters of this thesis are organized in a similar way and present contributions to mesh simplification followed by contributions to multi resolution techniques. Due to the large amount of works published so far the discussed algorithms are restricted to the most important works.

3.1 Mesh Simplification

The goal of automatic simplification is to generate an *optimal approximation* of a given complexity, that is, a mesh with a predefined number of mesh elements that minimizes an approximation error.

Starting with the pioneering work of Clark [Cla76], surface approximation algorithms have been developed in the eighties for restricted classes of surfaces with simple topologies like triangulated height fields or parametric surfaces. With the increasing facilities of range scanning systems and CAD systems in the early nineties, it has become possible to create huge polygonal models with very little effort demanding for algorithms to process such datasets. Automatic simplification techniques for more general surfaces have become important. At the same time, the marching cubes algorithm [LC87] established itself as a source of huge polygonal datasets.

Similar to polygonal surface models, the size of both structured and unstrucured volume datasets has started to grow with the increasing capabilities of mesh generation software [Rup95, She96], supercomputers and storage systems, so simplifications of such datasets have been demanded to reduce their sizes while approximating the dataset with a minimal approximation error.

3.1.1 Approximation Errors

A simplified mesh has an approximation error compared to the original mesh which is measured by an *error metric*. It is important to recognize that there is nothing like an universal error metric that is the best possible for all applications. Depending on the application type, a variety of different error metrics exist. The design of metrics that are appropriate for their applications and are computationally easy to evaluate is still an important part of simplification research.

Metrics for Triangular Meshes

For triangular meshes, the shape of the surface represented by the mesh must be well approximated. Many error metrics measure shape similarity as geometric deviation using Euclidean distances [GH97, LT99], parametric distances [COM98] or volume differences [LT98, SG98]. In many applications of computer graphics, simplified models are used to produce pixels in a final image. Here, metrics that account for the visual quality of a model are more appropriate like [LT00, KG00].

The Hausdorff distance is one of the most well-known metrics for making geometric comparisions between two point sets. It is based on another metric such as the Euclidean distance d_E between two points $\mathbf{p}$ and $\mathbf{q}$:

$$d_E(\mathbf{p}, \mathbf{q}) = \|\mathbf{p} - \mathbf{q}\|_2$$

The family of Hausdorff distances is defined as (see also figure 3.1)

- Hausdorff distance between a single point $\mathbf{p}$ and a point set Q:

$$d_S(\mathbf{p}, Q) = \inf_{\mathbf{q} \in Q} d_E(\mathbf{p}, \mathbf{q})$$

- (One-sided) Hausdorff distance between two point sets P and Q:

$$d_O(P, Q) = \sup_{\mathbf{p} \in P} \inf_{\mathbf{q} \in Q} d_E(\mathbf{p}, \mathbf{q}) = \sup_{\mathbf{p} \in P} d_S(\mathbf{p}, Q)$$

- (Two-sided) Hausdorff distance between two point sets P and Q:

$$d_H(P, Q) = \max(d_O(P, Q), d_O(Q, P))$$

As the one-sided Hausdorff distance is not a symmetric relation, it is not a metric in mathematical sense. On the other hand, the two-sided Hausdorff distance is symmetric and hence a metric.

Since Hausdorff distances are based on min/max functions of single point distances, they convey little information about the overall shape similarity. An alternative metric is based on the average distance between two surfaces P and Q

$$d_A(P, Q) = \frac{1}{|P|} \int_P d_S(\mathbf{p}, Q) d\mathbf{p}$$

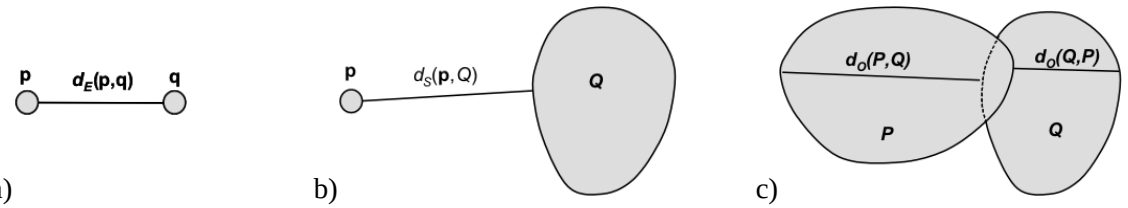


Figure 3.1: (a) The Euclidean distance between two points $\mathbf{p}$ *and* $\mathbf{q}$*. (b) The Hausdorff distance between a single point* $\mathbf{p}$ *and a point set* Q*. (c) The one-sided Hausdorff distances between to point sets* P *and* Q *are usually not symmetric, i.e.* $d_O(P,Q) \neq d_O(Q,P)$*.*

Here, $|P|$ denotes the area of surface P. Symmetry is obtained by taking the average of both asymmetric distances $d_A(P,Q)$ and $d_A(Q,P)$

$$d_M(P,Q) = \frac{|P|}{|P|+|Q|} d_A(P,Q) + \frac{|Q|}{|Q|+|P|} d_A(Q,P)$$

In the following, the metric d_M is referred to as *mean geometric error* whereas the two-sided Hausdorff distance d_H is referred to as *maximal geometric error*. Both metrics are implemented in the tools Metro [CRS98] and M.E.S.H. [ASCE02] which are public available. The maximal and the mean geometric error are designed for an offline comparision of two surfaces or curves and allow for a quality evaluation of simplification technqiues. But due to their high computational costs, they are often approximated during the simplification process itself.

Upper bounds for the maximal error d_H can be computed quickly like [Gui95, CMO97]. Here, the concept of simplification envelopes [CVM+96] is maybe the most notable one which explicitly computes and surrounds the model with two surface envelopes and restricts any simplified mesh element to lie within the envelopes.

The mean error d_M is often estimated locally [RR96] or by squared distances [HDD+93, Hop96, LT99]

$$\begin{aligned} d_S^2(\mathbf{p},Q) &= \inf_{\mathbf{q}\in Q} d_E^2(\mathbf{p},\mathbf{q}) \\ d_A^2(P,Q) &= \frac{1}{|P|}\int_P d_S^2(\mathbf{p},Q)d\mathbf{p} \\ d_M^2(P,Q) &= \frac{|P|}{|P|+|Q|} d_A^2(P,Q) + \frac{|Q|}{|Q|+|P|} d_A^2(Q,P). \end{aligned}$$

As a metric of particular interest not only for triangular but also for tetrahedral meshes, the *quadric error metric* [GH97] measures squared distances to a collection of triangle planes associated with each vertex as a approximation of the mean squared error. Its relevance is based on the important feature that the squared distance to any set of planes can be represented as a matrix which allows error computations to be extremely fast and optimal positions of new vertices to be computed by solving a system of linear equations. Additionally, surface attributes like color or texture coordinates can be naturally integrated and the metric can be extended to simplicial complexes of arbitrary dimension [GZ05].

Metrics for Tetrahedral Meshes

For tetrahedral meshes, two entities need to be approximated well: the underlying attribute field (comprising scalars and/or vectors) *and* the triangular boundary. While an *attribute error* measures the error of the attribute field, a *domain error* measures the geometric deviation of the boundary using any of the above geometric metrics. In the following, two tetrahedral meshes Σ_1 and Σ_2 are considered that discretize their domains Ω_1 and Ω_2 with the underlying attribute fields Φ_1 and Φ_2. Two types of attribute errors are distinguished, the field error and the space error.

The *field error* of a point $\mathbf{p} \in \Omega_1$ measures the absolute difference between the attribute value $\Phi_1(\mathbf{p})$ in one field and the attribute value $\Phi_2(\mathbf{p})$ in the other field

$$E^f(\mathbf{p}, \Omega_2) = \|\Phi_1(\mathbf{p}) - \Phi_2(\mathbf{p})\|.$$

The *maximal field error* is defined as the maximum of all field errors over the domain Ω_1

$$E_F^f(\Omega_1, \Omega_2) = \max_{\mathbf{p} \in \Omega_1} (\|\Phi_1(\mathbf{p}) - \Phi_2(\mathbf{p})\|)$$

Taking the maximal value does not reflect the volume size at which a field error happens. Slight changes in the shape of very small tetrahedra might result in a large maximal field error although the error occurs at a very tiny volume only and might be negligible. For this reason, the *mean field error* averages the field errors over the domain Ω_1

$$E_M^f(\Omega_1, \Omega_2) = \frac{1}{|\Omega_1|} \int_{\Omega_1} E^f(\mathbf{p}, \Omega_2) d\mathbf{p}$$

If both domains Ω_1 and Ω_2 do not exactly overlap, a point $\mathbf{p} \in \Omega_1$ may be outside of Ω_2, i.e. $\mathbf{p} \notin \Omega_2$. Here, the field Φ_2 cannot be evaluated directly but must be expanded to areas outside of Ω_2. For this, the point $\mathbf{p}$ is projected onto a point $\hat{\mathbf{p}}$ at the boundary of the closest cell (in Euclidean sense) and its field value is defined as $\Phi_2(\mathbf{p}) := \Phi_2(\hat{\mathbf{p}})$ [CCM+00].

The *space error* is the distance between a point $\mathbf{p}_1 \in \Omega_1$ with its attribute value $\phi = \Phi_1(\mathbf{p}_1)$ and the closest point $\mathbf{p}_2 \in \Omega_2$ that has the same attribute value ϕ. The field error is important for direct volume visualizations while the space error contributes to the quality of isosurface reconstruction.

Because of its computational cost, the field error is often approximated during simplification [SG98, THJW98, THJ99, KE00, CM02, CL03]. Additionally, the definition of an unified metric measuring both attribute errors and the domain error needs some care in order to avoid biasing one error type over the other. For simplification purposes, a weighted sum of addends is often defined where each addend either describes geometric or attribute field deformations. The weights must be chosen carefully in order to avoid biasing [CCM+00].

Due to their very general formulation, quadric error metrics [GZ05] have become a metric of particular interest for the simplification of tetrahedral meshes. The errors are measured as sum of squared distances to a set of hyperplanes associated with each vertex. The beauty of this approach is their ability to represent both attribute errors and the domain error of the boundary with a single $d \times d$ symmetric matrix and will be described in detail in § 3.1.4.

3.1.2 Simplification Strategies

Computing optimal approximations of height fields and by extension surface approximations is known to be NP-Hard [AS98]. Therefore, many existing simplification techniques are heuristic in nature and consist of an *atomic simplifying operation* together with an *error metric* describing the approximation error introduced by applying the operation. A mesh is then simplified by executing the atomic operation iteratively. The numerous approaches presented during the past years can be classified by their simplifying operations, see also figure 3.2. Note that the algorithms of chapter 4 cover technqiues of the types *Edge Collapse* and *Clustering of Vertices*.

Vertex Removal

A single vertex is removed from the mesh and the resulting hole is retriangulated. A hole in a triangular mesh can always be retriangulated which is not always possible in tetrahedral meshes. Here, some holes can only be tetrahedralized by adding so-called Steiner points which makes this approach pretty complex for tetrahedral meshes. For triangular meshes, popular approaches include vertex decimation [SZL92] and progressive encoding [AD01a]. Renze et al. have introduced vertex removal to unstructured tetrahedral meshes by their seminal paper [RO96].

Edge Collapse

The simplification operation contracts the two extreme vertices of an edge into a new point thereby removing all triangles (or tetrahedra) that are incident to the edge and deforming all triangles (or tetrahedra) incident to exactly one of both end vertices. Due to their nice features, edge collapses are todays standard in tetrahedral mesh simplification and will be described detailed in § 3.1.3. Among the numerous algorithms presented so far, Progressive Meshes [Hop96], quadric error metrics [GH97], half-edge collapse [KCS98], and memoryless simplification [LT98, LT99] are the most notable ones for triangle mesh simplification. [PH97] have extended Progressive Meshes to general simplicial complexes but do not take into account how an underlying attribute field of a tetrahedral mesh is approximated and have later been specialized to tetrahedral meshes.

[SG98] have presented one of the first works. The edge collapses are sorted by an error heap that uses a cost function which considers various errors like scalar field error or volume and shape deformation. The edge with the lowest cost is collapsed and the costs of all affected edges into the error heap are updated. This process is repeated until the heap is empty or a given number of collapses is performed. Additional tests are applied to check for mesh incosistencies like tetra folding, tetra intersection or tetra degeneration.

Independently, [THJ99] have also used a priority queue to simplify tetrahedral meshes by edge collapses. They have shown how an approximation of the error of an edge collapse can be easily guessed from the neighborhood of the edge. [CCM+00] have characterized the field and domain errors of an edge collapse. Various techniques are presented to evaluate or predict the field and domain error reliably and to prevent geometrical or topological degenerations like cell flipping

or self-intersections. Chiang et al. [CL03] have preserved the topological structure of isosurfaces of the mesh during simplification. Their algorithm first segments the mesh into topological-equivalent regions with all isosurfaces having the same topology within a region. Each region is then simpified independently by edge collapses thereby avoiding collapses that would cause a change of the isosurface topology.

[KE00] have simplified non-convex meshes and can change the topology of the mesh during simplification. In a preprocessing step, they add imaginery tetrahedra such that the mesh is transformed into a convex mesh. This mesh is simplified by sequential edge collapses. If an edge is collapsed that only belongs to imaginery tetrahedra, topology simplification is possible.

Finally, [GZ05] has extended quadric error metrics to tetrahedral meshes with any attribute values. Their approach is detailled in § 3.1.4.

Triangle / Tetrahedral Collapse

[Ham94] simplifies triangular meshes by iteratively collapsing triangles and removing all incident triangles. [THJW98] and [CM02] extend this concept to tetrahedral meshes where tetrahedra are iteratively collapsed. Because more triangles (or tetrahedra) are incident to a collapsing triangle (or tetrahedra) than to a collapsing edge, these simplification techniques are known to be fast but often result in approximations with a larger approximation error than methods based on edge collapses.

Clustering of Vertices

[RB93, LS01] determine spatial regions of a triangular mesh. All vertices within a region are collapsed into a single vertex which is positioned either at the midpoint of the region or at an error-minimizing location. Clustering approaches have been successfully used to rapidly reduce huge triangle datasets [LS01]. Their beauty comes from numerical robustness and ease of implementation. They have been extended to work out-of-core [Lin03]. They have found their way into tetrahedral mesh simplification recently by the independent work of the author [SS06a] and [UBF+05].

Approaches based on edge collapses play an important role because they are easy to implement and provide low approximation errors. A very fine control over the simplification process is achieved because just a single vertex is removed per operation. In contrast, the collapse of a whole triangle removes two vertices at once and may miss local situations which would better approximate the original mesh. Although a vertex removal also deletes just a single vertex, the advantage of edge collapses over vertex removals is that they can optimize the position of the new vertex with respect to an underlying error metric whereas vertex removals can only choose how the resulting whole

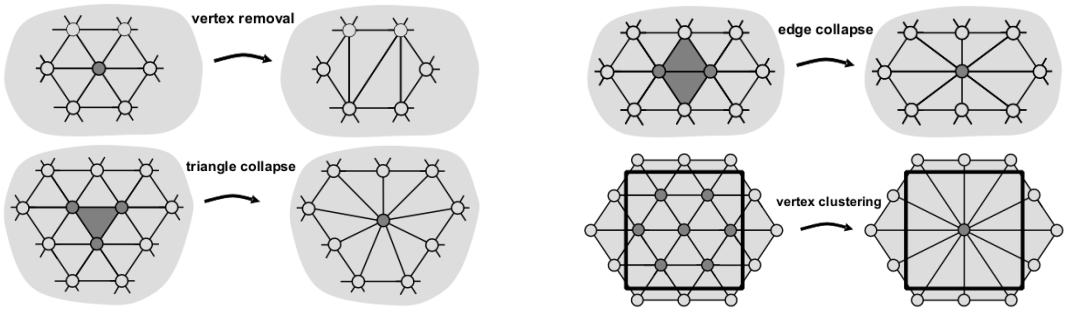


Figure 3.2: Simplification Operations: vertex removal, edge collapse, triangle collapse, and vertex clustering. The thick lines of the edge collapse operation are influenced edges whose error needs to be re-evaluated.

is retriangulated. Note that a retriangulation is by no means a trivial task if the number of poorly shaped triangles is to be minimized.

Another advantage of approaches based on edge collapses is their ability to create multi-resolution representations easily. Starting with the notion of *progressive meshes* [Hop96], each edge collapse records all information necessary to undo the collapse by recreating the edge and its incident triangles (or tetrahedra). Section 3.2 and chapter 5 describe such multi-resolution models in detail.

3.1.3 Iterative Edge Collapses

Because of their important ability to create high-quality approximations and to create multi-resolution models, I describe approaches based on edge collapses in more detail.

The simplification operation collapses both extreme vertices of an edge into a new single vertex deleting any mesh elements that are incident to the edge and deforming mesh elements that are incident to one of the two end vertices. Deformed mesh elements are called *modified mesh elements*.

A mesh M is simplified by applying a sequence of edge collapses $ecol_i$

$$M = M_n \overset{ecol_{n-1}}{\longrightarrow} M_{n-1} \overset{ecol_{n-2}}{\longrightarrow} \ldots \overset{ecol_0}{\longrightarrow} M_0.$$

There are two decisions involved with such iterative edge collapses. First, the position of the new vertex resulting from a collapse must be computed. Second, the order of the edge collapses must be determined. In general, both decisions depend on the chosen error metric. Beside simple approaches that place a new vertex at one of the extreme points of the edge, the new position is often chosen to minimize the error metric. Given the new position, the error metric determines the approximation error that an edge collapse introduces and provides the order of edge collapses from cheapest to costliest.

Traditionally, a queue keeps track of all $ecol$s and allows the edge with the lowest approximation error to be accessed in constant-time. After an $ecol$ has been applied, the approximation error of all modified edges must be re-estimated.

Many simplification algorithms based on edge collapses provide a zoo of methods to check if an edge shall be collapsed or not. The zoo includes methods to avoid the creation of badly shaped mesh elements, topology changes or geometric artifacts (like triangle or tetrahedral flips). For each edge $e = (p, q)$ that shall be collapsed, some of these preconditions can be checked:

- *Flipping.* If the volume of at least one modified triangle (or tetrahedron) changes its sign, the edge collapse is invalid.
- *Intersection.* If the mesh is non-convex, self-intersections may occur during an edge collapse at the mesh boundary. A rapid method for detection and treatment of self-intersections is presented in section 4.2.
- *Degeneration.* For the collapse of an edge $e = \{p, q\}$ to be topological valid, the following items need to be checked. Item (1) [DEGN99] is equivalent to the items (2,3,4) [HDD+93]

  1. For the collapse of an edge $e = (p, q)$ to be topological valid, the cut of the links of the extreme vertices must equal the link of the edge:
     $Lk(p) \cap Lk(q) = Lk(e)$
     This is true for any manifold mesh *without* boundary. Because mostly all tetrahedral meshes have a boundary, an additional dummy vertex needs to be added and connected to each triangle of the boundary via extra tetrahedra.
  2. For all vertices $\{r\}$ adjacent to both $\{p\}$ and $\{q\}$, $\{p, q, r\}$ is a face of the mesh.
  3. If $\{p\}$ and $\{q\}$ are boundary vertices, the edge $\{p, q\}$ is a boundary edge.
  4. The mesh has more then 4 vertices if neither $\{p\}$ nor $\{q\}$ are boundary vertices, or the mesh has more than 3 vertices if either $\{p\}$ or $\{q\}$ are boundary vertices (this is true for triangular meshes).

In addition to edge collapses, more general *vertex pair contractions* can be applied which merge two points into a new one. Both points need not to be connected by a mesh edge. So, the topology of the mesh can be modified and holes can be removed, for instance [GH97].

3.1.4 Quadric Error Metrics

A metric of particular interest for this work is the quadric error metric. It supports both triangular and tetrahedral meshes (and arbitrary simplicial complexes) in a uniform framework including any attribute values.

Historically, Ronfard and Rossignac [RR96] have attached a set of triangles to each vertex. Initially, the set of each vertex contains all its adjacent triangles. When both extreme vertices of an edge collapse into a new vertex, their triangle sets are merged into the triangle set of the new vertex whose position is chosen to minimize the sum of euclidean distances to all planes spanned by triangles in the new set. The maximal distance of the new position to all its planes defines the cost of the edge. Garland and Heckbert [GH97] have simplified this concept by using *squared*

distances which allow the formulation of distances to triangle sets as compact matrices. They called their metric quadric error metric. Later on, Garland and Zhou [GZ05] have shown that quadric error metrics can be extended to represent attributes very easily. Here, we follow the ideas of [GZ05] with slight modifications.

Fundamental Quadrics

Let's consider the d-dimensional space $\mathbf{R}^d$ spanned by an orthonormal basis $B = \{\mathbf{e}_i\}_{i=1}^d$. Due to the Pythagorean Theorem, the distance of a point $\mathbf{x} \in \mathbf{R}^d$ to another point $\mathbf{p} \in \mathbf{R}^d$ can be calculated as

$$\begin{aligned} \|\mathbf{x}-\mathbf{p}\|^2 &= \sum_{i=1}^{d}((\mathbf{x}-\mathbf{p})^T\mathbf{e}_i)^2 && (3.1)\\ &= \sum_{i=1}^{d}(\mathbf{x}-\mathbf{p})^T\mathbf{e}_i\mathbf{e}_i^T(\mathbf{x}-\mathbf{p}) && (3.2)\\ &= (\mathbf{x}-\mathbf{p})^T\left(\sum_{i=1}^{d}\mathbf{e}_i\mathbf{e}_i^T\right)(\mathbf{x}-\mathbf{p}) && (3.3) \end{aligned}$$

which directly implies $\sum_{i=1}^d \mathbf{e}_i\mathbf{e}_i^T = \mathbf{I}$ with the identity matrix $\mathbf{I}$ because we also know that

$$\|\mathbf{x}-\mathbf{p}\|^2 = (\mathbf{x}-\mathbf{p})^T\mathbf{I}(\mathbf{x}-\mathbf{p})$$

Hereby, the distance is uniformly measured with respect to all d basis vectors. Now consider a decomposition of the d-dimensional space $\mathbf{R}^d$ into two subspaces that are orthogonal to each other, the *tangent subspace* spanned by $B_T = \{\mathbf{e}_i\}_{i=1}^k \subseteq B$ and its so-called *normal subspace* spanned by $B_N = \{\mathbf{e}_i\}_{i=k+1}^d \subseteq B$, see also figure 3.3. If the error shall be measured to the normal subspace only, the identity $\mathbf{I}$ must be replaced by a matrix $\mathbf{A}$ with

$$\begin{aligned} \mathbf{A} &= \sum_{i=k+1}^{d}\mathbf{e}_i\mathbf{e}_i^T && (3.4)\\ &= \mathbf{I} - \sum_{i=1}^{k}\mathbf{e}_i\mathbf{e}_i^T && (3.5) \end{aligned}$$

The first equation allows $\mathbf{A}$ to be computed using all basis vectors that contribute to the distance whereas the second equation allows $\mathbf{A}$ to be computed using all basis vectors that do *not* contribute to the distance which is advantegous as will be described soon.

Note that $\mathbf{A}$ is a positive semi-definite matrix because an Euclidean distance is never smaller than zero. Putting $\mathbf{A}$ into equation (3.3), we get

$$\begin{aligned} \|\mathbf{x}-\mathbf{p}\|^2 &= (\mathbf{x}-\mathbf{p})^T\mathbf{A}(\mathbf{x}-\mathbf{p})\\ &= \mathbf{x}^T\mathbf{A}\mathbf{x} - 2\mathbf{p}^T\mathbf{A}\mathbf{x} + \mathbf{p}^T\mathbf{A}\mathbf{p}\\ &=: Q(\mathbf{x}) \end{aligned}$$

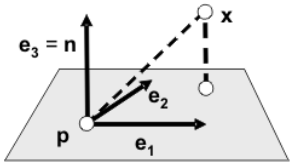


Figure 3.3: A plane is a two-dimensional subspace of R^3. The distance of a point $\mathbf{x}$ to the plane is measured within the one-dimensional normal subspace that is spanned by the plane's normal vector $\mathbf{n}$.

which is a quadric form and can – for clearness – be written as

$$Q(\mathbf{x}) = \mathbf{x}^T\mathbf{A}\mathbf{x} + 2\mathbf{b}^T\mathbf{x} + c$$

where

$$\mathbf{A} = \mathbf{I} - \sum_{i=1}^{k} \mathbf{e}_i\mathbf{e}_i^T, \quad \mathbf{b} = -\mathbf{A}\mathbf{p}, \quad c = \mathbf{p}^T\mathbf{A}\mathbf{p}.$$

$\mathbf{A}$ is a $d \times d$ symmetric, positive semi-definite matrix and can be represented with $\frac{1}{2}d(d+1)$ floating point values, $\mathbf{b}$ is a d-vector and c is a scalar. So, a total of $\frac{1}{2}d(d+3)+1$ floating point values are sufficient to represent Q as a triple

$$Q = (\mathbf{A}, \mathbf{b}, c) \tag{3.6}$$

Q is called a *quadric*. Up to now, no assumptions have been made about the d-dimensional space $\mathbf{R}^d$. Due to this very general formulation, $\mathbf{R}^d$ is not restricted to cover geometrical (i.e. positional) information alone. By increasing dimensionality, additional attribute information can be integrated. Given vertex positions $\mathbf{x} = (x_1, ..., x_D) \in \mathbf{R}^D$ and additional attribute values $\mathbf{a} = (a_1, ..., a_A) \in \mathbf{R}^A$ at each vertex, quadrics measure squared distances within the $d = (D + A)$-dimensional space $\mathbf{R}^{D+A}$ consisting of all concatenated vectors

$$\mathbf{x} = (x_1, ..., x_D, a_1, ..., a_A)^T \in \mathbf{R}^{D+A}$$

.

The choice of the orthonormal basis B is given by the application. For triangular meshes embedded in $\mathbf{R}^3$ without attribute values, an orthonormal coordinate system is established for each triangle comprising the normal vector $\mathbf{n} = \mathbf{e}_3$ that spans the normal subspace and two vectors $\mathbf{e}_1$ and $\mathbf{e}_2$ that span the tangent space, see figure 3.4. Given an *anchor point* $\mathbf{p}$ on the triangle t, equation (3.4) leads to the well-known quadric formula for single triangles

$$Q_t(\mathbf{x}) = (\mathbf{x} - \mathbf{p})^T(\mathbf{n}\mathbf{n}^T)(\mathbf{x} - \mathbf{p})$$

The quadrics Q_t for single triangles (or tetrahedra) are called *fundamental quadrics*. If the triangular mesh carries A attribute values like colors or texture coordinates at its vertices, the

dimensionality increases to $\mathbf{R}^{3+A}$ with $(3 + A)$ basis vectors. Here, equation (3.5) allows for a simple computation of the quadric using the basis vectors $\mathbf{e}_1, \mathbf{e}_2 \in \mathbf{R}^{3+A}$. Given a triangle $t = (\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3)$ with $\mathbf{x}_i \in \mathbf{R}^{3+A}$, the basis vectors are computed by Gram-Schmidt orthogonalization of the edge vectors $\hat{\mathbf{e}}_1$ and $\hat{\mathbf{e}}_1$ as visualized in figure 3.4a:

$$\begin{aligned} \hat{\mathbf{e}}_1 &= \mathbf{x}_1 - \mathbf{x}_0 \\ \hat{\mathbf{e}}_2 &= \mathbf{x}_2 - \mathbf{x}_0 \end{aligned}$$

resulting in the quadric Q_t for triangle t

$$\mathbf{A} = \mathbf{I} - \mathbf{e}_0\mathbf{e}_0^T - \mathbf{e}_1\mathbf{e}_1^T, \;\; \mathbf{b} = -\mathbf{A}\mathbf{p}, \;\; c = \mathbf{p}^T\mathbf{A}\mathbf{p}.$$

For tetrahedral meshes with attribute values at its vertices, the basis vectors of the tangent space can be computed from the three edge vectors of a tetrahedron using Gram-Schmidt orthogonalization as shown in figure 3.4b. The quadric Q_t for a tetrahedron t is simply

$$\mathbf{A} = \mathbf{I} - \sum_{i=0}^{2} \mathbf{e}_i\mathbf{e}_i^T, \;\; \mathbf{b} = -\mathbf{A}\mathbf{p}, \;\; c = \mathbf{p}^T\mathbf{A}\mathbf{p}. \tag{3.7}$$

with $\mathbf{e}_i \in \mathbf{R}^{3+A}$. The anchor point $\mathbf{p}$ is chosen as the center of a triangle or the center of a tetrahedron, respectively.

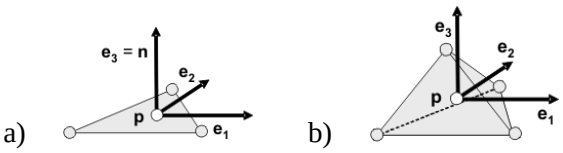


Figure 3.4: (a) The tangent subspace is a plane as given by the triangle. (b) The tangent subspace is formulated as the subspace comprising the tetrahedron itself as well as all attribute values at its four vertices.

Quadrics for Vertices

A fundamental quadric measures the squared distance of a point to a single subspace spanned by a triangle or tetrahedron. When quadrics are added, the new quadric describes the sum of squared distances to a set of subspaces. Adding quadrics simply means to sum the components of the quadrics (3.6).

Given the fundamental quadrics Q_t of triangles or tetraehedra, a quadric for a vertex is computed by adding the fundamental quadrics of all its incident triangles

$$Q_v = \sum_{f \in St(v)} w_t Q_t.$$

The weights w_t steer the influence of a fundamental quadric. In order to achieve scale-independent quadrics, w_t is usually set to the area of its triangle t or the volume of its tetrahedron t.

If two vertices p and q collapse into a new vertex v, the new quadric is just the sum of the quadrics of both vertices $Q_v = Q_p + Q_q$ measuring the sum of squared distances to all subspaces given by the addends.

The position of a new vertex v is chosen to minimize its quadric Q_v. As a necessary condition, the first deravative must be zero at a minimal value $\mathbf{x}_v^*$:

$$\nabla Q_v(\mathbf{x}_v^*) = 2\mathbf{A}\mathbf{x}_v^* + 2\mathbf{b} = 0$$

The resulting linear system

$$\mathbf{A}\mathbf{x}_v^* = -\mathbf{b}$$

can be solved using Cholesky decomposition [GZ05], conjugate gradient iterations [VCL+05] or even SVD decomposition [LS01] because $\mathbf{A}$ is symmetric and positive semi-definite. If conjugate gradient iterations are used, the middle point of the collapsing edge is chosen as starting point.

As one of the most beautiful properties of quadric error metrics, the matrix formulation allows quadrics to be added together even if they measure squared distances to different subspaces. So, different error types can be described by a single quadric.

For example, the tetrahedral quadrics of equation (3.7) approximate the squared mean field error but do not consider the domain error. They measure the squared distance within the A-dimensional normal space of $\mathbf{R}^{3+A}$ which is a hyperplane for $A = 1$. This leads to distorted boundaries, i.e. large domain errors. If the boundary of the mesh shall be approximated well, so-called *face quadrics* [GZ05] can be added which are basically triangle quadrics for boundary triangles penalizing the movement of vertices across the boundary.

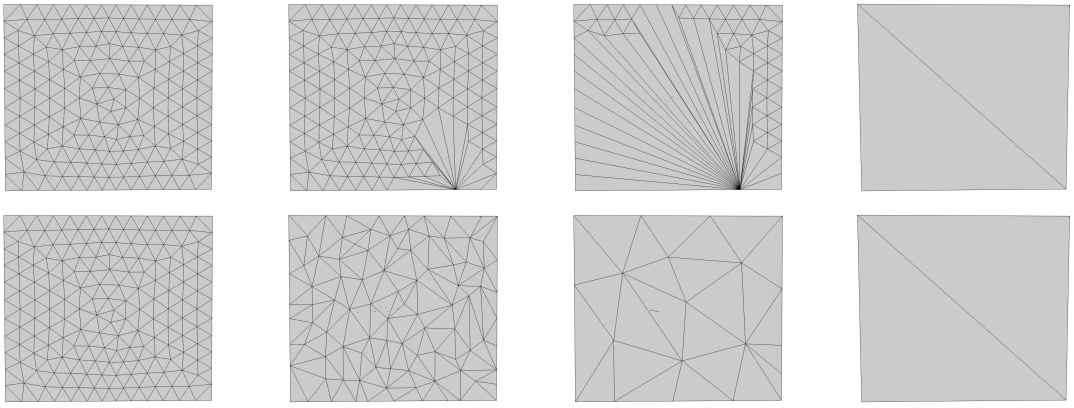


Figure 3.5: Top row: without vertex distribution quadrics. Bottom row: with vertex distribution quadrics.

As another example, in degenerated parts of meshes like flat triangle meshes or tetrahedral meshes without attributes, the matrix $\mathbf{A}$ is ill-conditioned or even not invertible. Beside a solver

for linear systems which is numerical robust [VCL+05, LS01], additional quadrics can be added to each vertex in order to increase the condition of $\mathbf{A}$. So, Garland and Zhou [GZ05] suggest to add *vertex distribution quadrics* of the form

$$\mathbf{A} = \mathbf{I}, \quad \mathbf{b} = -\mathbf{p}, \quad c = \mathbf{p}^T\mathbf{p} \tag{3.8}$$

to each vertex $\mathbf{p}$. They penalize the movement of vertices and bias a quadric to preserve existing vertex distributions, see figure 3.5. Such vertex distribution quadrics raise the condition of $\mathbf{A}$ and tend to produce well-shaped triangles but assume that a mesh is uniformly sampled. Although this is often the case for polygonal models, it is often not the case for triangle meshes that bound tetrahedralized volumes. Here, the point distribution can vary strongly and vertex distribution quadrics must be weighted carefully.

3.2 Multi-Resolution Models

The seminal paper of Clark [Cla76] has introduced a hierarchy of multiple representations with different complexity for each polygonal object in a scene. An appropriate level of detail can be chosen for each object at run time in order to elevate display rates. Based on this work, a variaty of subsequent algorithms have been published. I want to cover the most important ones for both triangular and tetrahedral meshes. But let's start with a general overview to the concepts of multi-resolution models [CDFL+04].

3.2.1 Basic Concepts

A *multiresolution model* M can be considered as a triple $(\Sigma_0, \mathcal{U}, \mathcal{D})$. Σ_0 is the coarsest approximation of the original mesh and is called the *base mesh*, $\mathcal{U}$ is a list of *updates*, and $\mathcal{D}$ describes a direct *dependency* relationship for the updates.

An update modifies a mesh by removing and inserting mesh elements like vertices, edges, triangles, or tetrahedra. An update $u \in \mathcal{U}$ can be considered as a pair $u = (u^+, u^-)$ with u^- as a coarsening modification and u^+ as a refining modification. Often (but not always), the base mesh Σ_0 has been built by applying a sequence of coarsening updates u^- to the original mesh Σ which is also called *reference mesh*.

The dependency relationship $\mathcal{D}$ defines dependencies between updates $u \in \mathcal{U}$. An update u_1 depends directly on another update u_2, if u_2^+ removes mesh elements that have been introduced by u_1^+. Hence, the update u_2^+ can only be applied if u_1^+ has been applied before. It can be shown that this relationship defines a partial order on the updates [Pup98], so $\mathcal{D}$ can be described as a directed acyclic graph (DAG).

If the update u uses edge collapses as coarsening operation u^- and vertex splits as refining operation u^+, u^- removes all modified triangles (tetrahedra) and replaces them by new triangles (tetrahedra) as shown in figure 3.6.

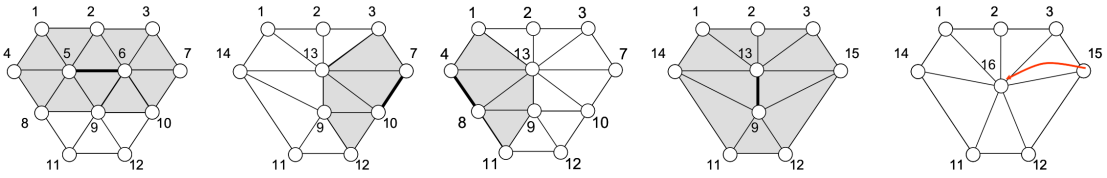


Figure 3.6: A triangle mesh is simplified by a sequence of edge collapses from left to right. Collapsed edges are shown as bold lines. The corresponding vertex splits run from right to left. Each (vertex split, edge collapse)-pair forms an update $u = (u^+, u^-)$. All triangles that are influenced by an edge collapse are gray. Note that an vertex split can be performed only if all those triangles exist that have been existed at the time of the corresponding edge collapse. For instance, vertex 15 can be split only if vertex 16 is split before.

Traditionally, a multi-resolution model is constructed in a preprocessing step. The original (reference) mesh Σ is simplified down to a base mesh Σ_0 recording the update list $\mathcal{U}$ together with all dependencies $\mathcal{D}$. At run time, a mesh conformal to both the base mesh and the reference mesh is created by applying updates in an order defined by their dependencies to the base mesh. For an elevated performance it is not necessary to start the refinement at the base mesh every time. A conformal mesh can be created incrementally from another conformal mesh by applying a specific subset of the updates and considering their dependencies.

Finally, many algorithms for general meshes fall into one of two categories: they either work on the *vertex level* or on the *segment level*. Vertex-level Algorithms use the operations vertex split and edge collapse while segment-level algorithms replace a part of the mesh, i.e. a segment, by coarser or finer parts, i.e. other segments. This thesis covers algorithms of both categories. While the presented vertex-level algorithms of chapter 5 improve known techniques in terms of space and speed, the segment-level algorithms of chapter 6 are the first ones ever presented in the area of tetrahedral meshing. Chapter 5 also discusses the differences between previous works on vertex-level algorithms and the new algorithms presented there.

3.2.2 Triangular Meshes

Many early multi-resolution models have been designed for surfaces with a limited complexity like triangulated height fields [FL79] and have later been improved to more sophisticated models like BDAM [CGG$^+$03], Geometry Clipmaps [LH04], and RUSTIC [Pom00] or CABTT [Lev02] as extensions of ROAM [DWS$^+$97].

Multi-resolution models for general triangular meshes in 3D have started to show up in the mid nineties with the increasing size of available polygonal meshes. Taking the progressive representation of a mesh as starting point [Hop96], many early models have been based on a hierarchy of possible refinement/coarsening operations at the vertex or triangle level. Xia and Varshney [XV96] and Hoppe [Hop97] use edge collapses and vertex splits whereas El-Sana and Varshney [ESV99] use vertex clustering. Lindstrom [Lin03] has proposed a scheme for out-of-core construction and

visualization of multi-resolution surfaces based on vertex clustering on a rectilinear octree.

As a main contribution, [XV96] encode the dependencies $\mathcal{D}$ between updates as a forest of binary trees of vertices and a vertex enumeration scheme as shown in figure 3.7. The leaves of the forest correspond to the vertices of the reference mesh while the interiour nodes correspond to all vertices introduced by edge collapses. If an edge $e = \{p, q\}$ collapses into a vertex v, then p and q are the children of v.

The n vertices of the reference mesh can be enumerated in any order from 1 to n. The remaining vertices are enumerated in ascending order as defined by the sequence of edge collapses. The dependencies are given implicitly by the vertex enumeration. The nice property of this enumeration is that an update u_1 depends on another update u_2 if the number of its corresponding vertex in the hierarchy is lower than the number of u_2. Because this enumeration scheme is important for my multi-resolution structure, it is described in more detail in § 5.2.

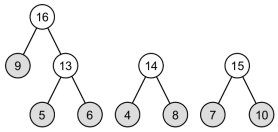


Figure 3.7: The forest of binary trees forms a vertex hierarchy corresponding to the sequence of figure 3.6. Gray circles are original vertices. The vertices that are introduced by edge collapses are enumerated in ascending order.

Starting with Adaptive TetraPuzzles [CGG+04], the hierarchies are constructed with possible refinement/coarsening operations at a segment level. Each segment – or patch – contains small parts of the model and can be replaced by coarser or finer segments. Adaptive TetraPuzzles uses a conformal hierarchy of tetrahedra generated by recursive longest edge bisection to spatially partition the model. Each cell contains a precomputed simplified version of the original model. The representation is constructed off-line during a fine-to-coarse parallel out-of-core simplification of the surface. Appropriate boundary constraints are introduced in the simplification process to ensure that all conforming selective subdivisions of the tetrahedra hierarchy lead to correctly matching surface patches.

A related approach has been presented in the QuickVDR system [YSGM04]. The original model is paritioned into a hierarchical set of small patches that are independently simplified into progressive meshes and merged bottom-up. Additional logic in the management of boundaries between clusters allows patch boundaries to be simplified while enforcing the conformality of the resulting mesh.

Batched Multi-Triangulations [CGG+05] extend the concept of multi triangulations [Pup98] from the vertex level to the patch level. The model is partitioned into patches which are independently simplified. Next, a *different* partition is chosen based on the simplified patches and the resulting new patches are again independently simplified. A hierarchy is recorded as a directed acyclic graph whose arcs encode which patches of different levels overlap. The graph enables a

conformal mesh to be assembled as a list of non-overlapping patches without gaps. Because this work has inspired my work on tetrahedral meshes, the concepts of multi-triangulations are detailed in § 6.3.

3.2.3 Tetrahedral Meshes

Previously published multi-resolution models for unstructured tetrahedral meshes rely on the refinement / coarsening operations edge collapse / vertex split at the vertex level. Although rather general descriptions exist that describe how a patch-based model can be applied, I have been the first to introduce a working multi-resolution model which is specialized for huge tetrahedral meshes and works at the patch level.

In essence, many vertex-based multi-resolution models are extensions of their triangular counterparts and exploit the vertex enumeration scheme of [XV96] in order to ensure conforming meshes. The main difference to models for triangular models is how the update information for a vertex split is encoded. A vertex split partitions the tetrahedra incident to a split vertex v_s into two sets which are separated by a fan of triangles incident at v_s. Tetrahedra of the two sets are deformed to become incident to v_a and v_b, respectively. All triangles belonging to the fan become tetrahedra to both v_a and v_b as shown in figure 3.8. In particular, the challenge of vertex splits for tetrahedral meshes is that tetrahedra around a split vertex do not have a canonical ordering, that is, there is nothing like a clock-wise or counter-clock-wise traversal possible as it is frequently used in triangle meshes [Hop97] in order to find both sets of triangles.

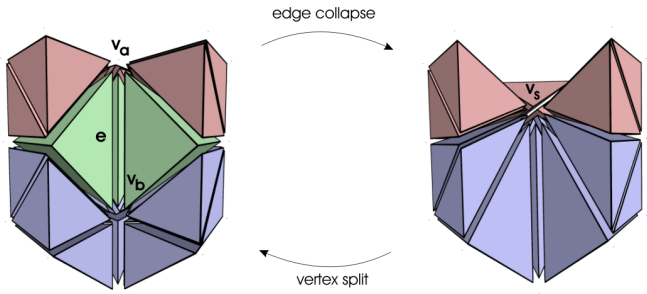


Figure 3.8: An edge collapse and vertex split modifies a tetrahedral mesh. An edge collapse removes all tetrahedra incident to the edge while a vertex split must partition the surrounding tetrahedra into two sets in order to re-insert all collapsed tetrahedra between both sets.

Cignoni et al. [CMRS03] introduce a multi-resolution model based on edge collapses. A sequence of edge collapses constructs a binary hierarchy of vertices in a preprocessing step where each update u of the hierarchy stores

- *Geometry.* Offset vectors which are used to find the positions of v_a and v_b from that of v_s, and vice-versa.

- *Attributes.* Offset values which are used to find the field attributes of v_a and v_b from that of v_s, and vice-versa.
- *Error.* An error value providing an estimate of the field error E^f associated with u.
- *Connectivity.* A bit mask which is used to find the partition of all tetrahedra incident to v_s.

The bit mask contains a single bit for each tetrahedron incident to v_s. In order to establish an order, all tetrahedra incident to v_s are sorted lexicographically by the triplets formed by their vertices different to v_s. Each triplet itself sorts its vertices according to their indices. Given the order of tetrahedra and the bit field, all tetrahedra marked with 0 replace v_s with v_a while tetrahedra markd with 1 replace v_s with v_b. Each triangular face shared by tetrahedra with different marks must be expanded into a tetrahedron incident to both v_a and v_b.

The most interesting storage cost is the size of the bit field which is restricted to 64 bits or 8 bytes and makes up the half storage cost of all update information.

Danovaro et al. [DDF02, DDFM$^+$05] introduces a multi-resolution model based on half-edge collapses where an oriented edge $e = (v_a, v_b)$ collapses into its extreme vertex v_a. Similar to [CMRS03], a binary hierarchy of vertices is constructed. Because half-edge collapses do not create new vertices, a vertex labelling scheme is used to enumerate vertices in ascending order and to ensure conformal meshes by again exploiting the vertex enumeration scheme of [XV96].

Similar to [CMRS03], geometry information, attribute information, field error information and connectivity information are stored. The latter allows for a different representation due to the restriction to half-edge collapses. With each update u, the local index of a tetrahedron τ within all tetrahedra incident to v_s is stored together with a code for the starting face f within τ and a bit stream describing a traversal of all tetrahedra incident to v_s starting at the face f.

The local index and the face need a total storage of 5 bytes and the bit stream requires additional 4 bytes summing up to 9 bytes. But the bit stream does not need a special sorting of incident tetrahedra but encodes a unique traversal starting at the face f within the tetrahedron τ.

In addition to multi-resolution models, Cignoni et al. [CDFM$^+$04] present a query scheme for run–time adaptation of a multi-resolution meshes which is based on two queues [DWS$^+$97] and enables a mesh to adapt to given viewing or classification parameters very detailled. The view–dependent parts of my own research in chapter 5 was mainly influenced by these works.

There are several techniques specialized for tetrahedral meshes with subdivision connectivity that are not covered here. Additionally, some multi-resolution approaches subdivide regular grids by tetrahedra in order to adapt a tetrahedral mesh to an isosurface. Note that there is a big difference between such models and models for unstructured tetrahedral meshes.

Chapter 4

Mesh Simplification

Everything should be made as simple as possible, but not simpler. - Albert Einstein

As the size of geometric datasets has continued to grow very rapidly over the last years, automatic simplification techniques have become an important tool for managing this huge amount of data. The original dataset is simplified to an approximating dataset with far fewer data elements. A whole bundle of applications ranging from computer graphics and visualization to simulation software for finite elements benefits from such tools. Surface models produced by scanning and reconstruction systems are almost always highly oversampled. Simulation systems run their code on supercomputers and produce huge volumes of output data that are far to big to be visualized interactively because visualization systems often have only commodity PC hardware available.

4.1 Iterative Edge Collapses Revisited

This section covers two minor issues for edge collapse based simplifiers.

Consider an edge $e = \{p, q\}$ in a triangular mesh with extreme vertices p and q as shown in figure 4.1. We examine the valences of p and q together with the valences of their wing vertices, that is, the valences of the (at most) two vertices that are incident to both p and q. The valences of both vertices are $V(p)$ and $V(q)$, respectively. If the edge collapses into a new vertex v, the valence of v is simply

$$V(v) = V(p) + V(q) - 4. \tag{4.1}$$

So, the valence of v is larger than the valences of p and q if $V(p) + V(q) > 4$. On the other hand, the valence of each wing vertex decreases by 1 because the two edges connecting the wing vertex to p and q are replaced by a single edge connecting the wing vertex to the new vertex v.

Similar, if an edge $e = \{p, q\}$ collapses in a tetrahedral mesh, the valence of a new vertex v is given by

$$V(v) = V(p) + V(q) - 2 - 2k + k = V(p) + V(q) - 2 - k, \tag{4.2}$$

where k is the number of vertices incident to both p and q. Again, the valence of v is larger than the valences of the collapsing vertices if $V(p)+V(q) > 2+k$. Note that equation (4.1) is a special case of equation (4.2) with $k = 2$.

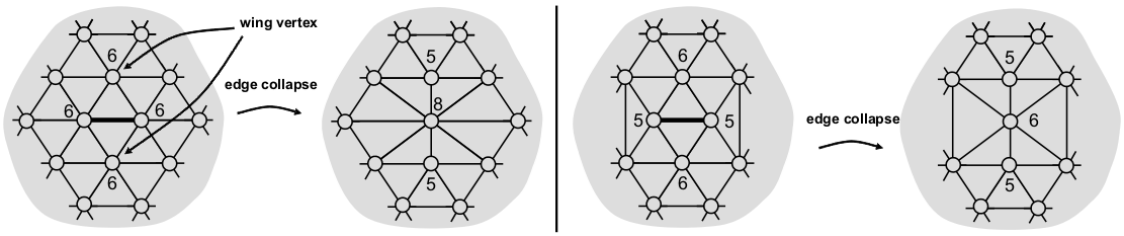


Figure 4.1: The valence of the new vertex increases while the valences of the wing vertices decreases.

In summary, an edge collapse leaves a single vertex with high valence and k wing vertices with low valence. Together with the Euler formula, the distribution of valences spreads increasing both the number of high-valence vertices and low-valence vertices. But a vertex with a high valence causes its incident triangles (or tetrahedra) to be poorly shaped while valence-3 vertices tend to invalidate future edge collapses due to topological restrictions.

The following section describes a simplifier which controles the valences improving both the robustness of the simplification process itself and the quality of multi-resolution representations. Simplifiers frequently compute volumes of tetrahedra which is ill-conditioned for badly shaped tetrahedra at high-valence vertices. The number of dependencies between updates is decreased in a multi-resolution mesh because a vertex may depend directly on all its incident vertices. The number of incident vertices is controlled by controlling the valences.

4.1.1 Independent Edge Collapses

This section describes a simplification method that considers the vertex valence distribution by collapsing independent edges only. It does not store all edges in a queue but just a workload of edges which typically contains much less edges. It is not only suitable for a quick and robust simplification with errors similar to approaches based on queues that sort all edges but also for the construction of multi-resolution models due to its concept of independent edges.

Two edges are *independent* of each other if they are incident to different vertices as visualized by figure 4.2. The algorithm simplifies a mesh in several runs over the mesh. Each run detects a set of independent edges which are collapsed. This way, the mesh is uniformly simplified which can be perfectly used for a progressive representation as described in § 5.2.

A single run starts at an arbitrary edge which is collapsed into a new vertex v. All edges in the link of the new vertex v are added to a queue, i.e. all edges $e \in Lk(v)$. Next, the next best edge from the queue is collapsed into $\hat{v}$, removed from the queue and all edges in the link of $\hat{v}$ are put onto the queue if they are independent of previously collapsed edges.

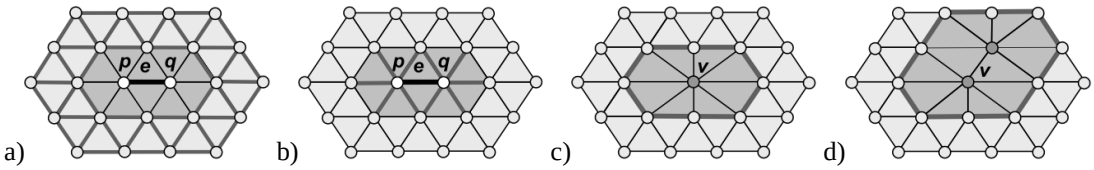


Figure 4.2: (a) Edges independent of edge e are dark gray. (b) Dependent edges are dark gray. (c) After an edge collapse, all edges in the link of v, $Lk(v)$, are put onto the queue. (d) The next simplification step increases the list of queued edges.

The independence of edges to previously collapsed edges is ensured by marking vertices. Initially, all vertices are unmarked. When an edge collapses into a new vertex v, v is marked. An edge is independent of another previously collapsed edge (and can be added to the queue) if and only if both its extreme vertices are not marked.

The queue sorts all edges by their quadric costs. When an edge collapses, it increases the valence of the new vertex but decreases the valences of wing vertices. But because only independent edges are put on the queue containing edges that are incident to the wing vertices, those wing vertices are likey to be collapsed next thereby *increasing* their valence again and *decreasing* the valence of their wing vertices, respectively. This way, valences do not spread because low-valence wing vertices are collapsed decreasing the valence of their wing vertices which are likely to be the high-valence vertices left by previous edge collapses.

Due to the selection of independent edges, edges may be collapsed with a high error. In order to prevent the collapse of such high-error edges, a threshold is introduced. The initial value of the threshold is estimated by selecting k edges randomly and setting the threshold to the average quadric cost of all k edges. k is typically set to 10. This process is repeated before every run over the mesh.

Name	Sea	Post	Fighter	F16
# Vertices	102,165	108,300	256,614	1,124,648
# Tetra	524,640	624,153	1,403,504	6,345,709
Reduction to	10%	10%	10%	5%
$\tilde{E}_M^f$	0.12	0.12	0.009	0.08
Time [sec]	27	76	189	781
Max Queue Size [# Edges]	48,312	82,591	105,195	341,629
$\tilde{E}_M^f$, no IEC	0.09	0.10	0.007	0.05
Time [sec], no IEC	34	98	210	923
Max Queue Size	655,228	740,850	1,701,869	7,625,318

Table 4.1: The mean field errors $\tilde{E}_M^f$ and timings of independent edge collapses compared to our simplifier with full queues. Timings are given without file I/O.

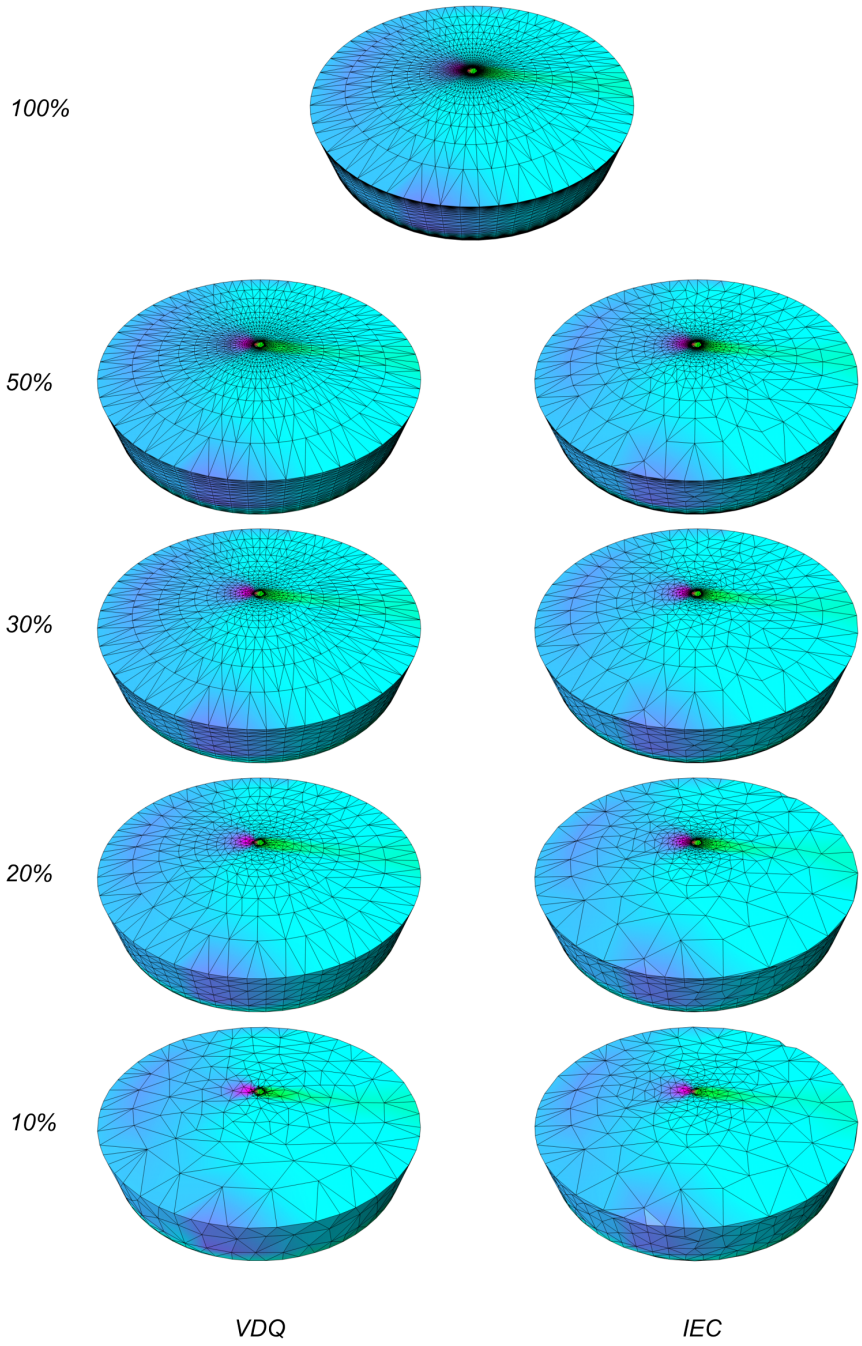


Figure 4.3: The tetrahedral Post mesh is simplified from the full mesh (100%) to 10%. VDQ = with vertex distribution quadric, IEC = independent edge collapses.

4.1.2 Feature Edges

Consider a triangle mesh with a very sharp edge as shown in figure 4.5. Such edges do typically not appear in triangle meshes describing 3D surface models but appear frequently in triangle meshes that bound volumetric tetrahedral meshes like the mesh at the tip of an aircraft's wing. Here, they model important features of the mesh and should be approximated well in a simplified mesh. Nevertheless, the condition of the quadric's matrix $\mathbf{A}$ is very low and needs to be improved by adding additional error terms.

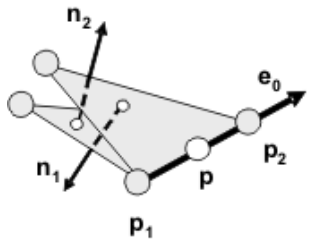


Figure 4.4: An edge quadric is defined along a sharp edge elevating the condition of the linear system.

The quadrics for tetrahedral meshes steer the optimal position to minimize the scalar field and do not take actual geometry into account which means that a vertex is likely to be moved away from the edge. If additional face quadrics are added, the minimization problem gets ill-conditioned because the triangle border faces are nearly parallel to each other while having normals that point into opposite directions.

In order to avoid these problems, we add an *edge quadric* which makes the linear system well-conditioned by penalizing movements across the edge. Sharp edges are identified by a threshold on the bending angle along the edge. Note that an edge quadric measures distances almost orthogonal to a face quadric. Given an edge with end vertices $\mathbf{p}_1$ and $\mathbf{p}_2$ as shown in figure 4.4, the edge quadric is defined along the edge

$$\begin{aligned} \mathbf{e}_0 &= \frac{\mathbf{p}_2 - \mathbf{p}_1}{|\mathbf{p}_2 - \mathbf{p}_1|} \\ \mathbf{A} &= \mathbf{I} - \mathbf{e}_0\mathbf{e}_0^T, \quad \mathbf{b} = -\mathbf{p}, \quad c = \mathbf{p}^T\mathbf{p}. \end{aligned}$$

Similar to face quadrics, all edge quadrics are weighted by a user-given factor w_e steering the influence to the approximation as shown in figure 4.5. The beauty of this approach arises from its general application to both triangle and tetrahedral meshes.

4.2 Vertex Clustering

As described above, approaches based on edge collapses are known to create high-quality approximations especially if quadric error metrics are used [GZ05]. Nevertheless, they need to check

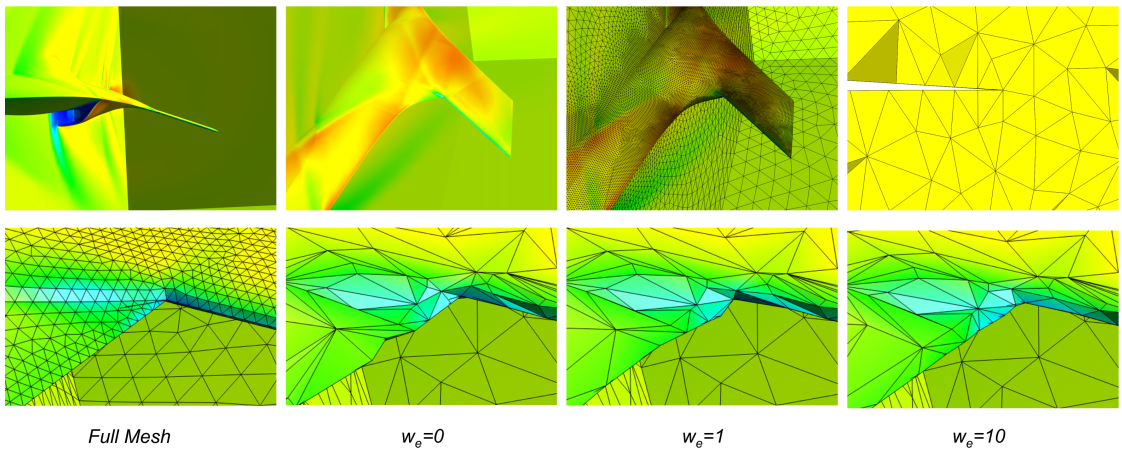


Figure 4.5: Top row: The edge at the wing tip of the Fighter model is highly detailed. The border triangular mesh is rendered from inside the dataset looking at the edge. Top right: Tetrahedra encapsulate the wing and leave just a small gap (which models the wing itself). The bottom row shows a zoom onto the wing tip for different edge weights w_e.

whether an edge collapse is valid and thus need to compute volumes of tetrahedra which is ill conditioned for badly shaped tetrahedra (e.g. slivers). Additionally, if the data structures support manifold meshes only, additional care must be taken to ensure that an edge collapse doesn't introduce non-manifold vertices. Furthermore, a sequence of edge collapses is a sequence of local operations. Thus, the simplifier may run into a local minimum without reaching a global minimum.

The simplification algorithm of this section overcomes these problems by looking globally at the tetrahedral mesh. Instead of using a series of local operations that need to be valid, a sequence of global operations is performed which avoids the problems of local operations.

First, the border of the tetrahedral mesh (which is a triangle mesh) is simplified into an intersection-free triangle mesh that approximates the original border. Second, we locate all points that shall be contained in the simplified tetrahedral mesh by point sampling the interior of the tetrahedral mesh. Third, we compute a Delaunay tetrahedralization of the sample points which conforms to the simplified border. The result is a tetrahedral mesh that contains all sample points (plus so-called Steiner points) and is bounded by the simplified border. The approach is extremely fast, simple to implement, numerical stable and cannot run into local minima. Furthermore, it provides a clear separation of boundary simplification and attribute field simplifications (see section 3.1.1).

Independently, Uesu et al. [UBF+05] designed a similar system. We outrival their algorithm by two important points. First, we ensure that the simplified border is *intersection-free*. A tetrahedralizer can't constrain a tetrahedral mesh to self-intersecting faces. In most cases, it just crashes. The intersection-free border simplification was a major factor during the development of our algorithm because some of our test meshes are likely to produce self-intersections. Second, our point

sampling takes not only an *field error* into account but also a *space error* which is very important for isosurface reconstruction. So, our reconstructed isosurfaces have a much lower distortion than [UBF+05]. Furthermore, our point sampling is more regular which results in better-shaped tetrahedral meshes and faster run-times for the tetrahedralizer.

As the first step of our tetrahedral simplifier, we show how a surface triangle mesh can be simplified to an intersection-free approximation. The major challange is to reject as early as possible as many triangles as possible such that the number of expensive geometric intersection tests can be minimized.

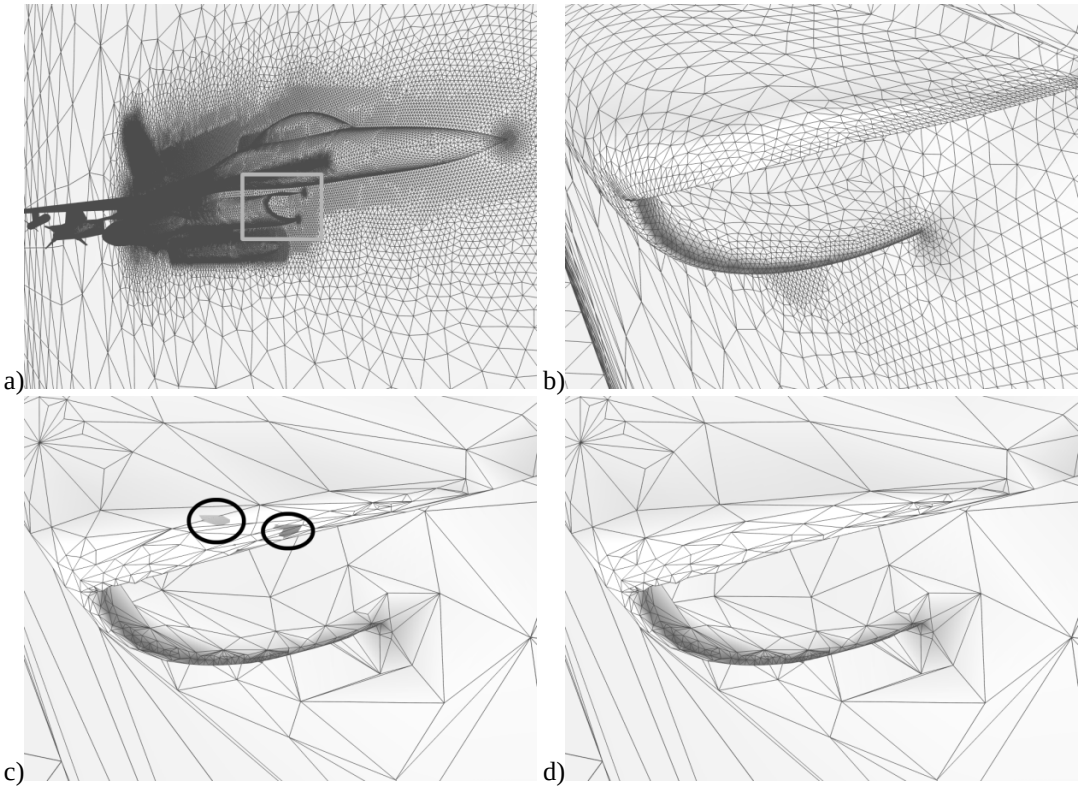


Figure 4.6: (a, b) The tetrahedral mesh of a F16-like fighter aircraft contains more then six million tetrahedra. Only the border triangles are shown as they are seen from inside the tetrahedral mesh. The border mesh is highly detailled with many faces that are nearly parallel but a short distance away from each other. (c, d) Intersections during the border simplification are likely (c, see circles) and need to be avoided (d).

We embedded this intersection detection into a traditional edge collapse approach for triangle meshes with quadric error metrics, see section 4.1. Typically, each vertex stores a quadric which measures the quadric distance of the vertex to the faces of the original mesh. Our approach stores additionally a region of influence (i.e. a bounding box) for each vertex that covers all incident triangles of the vertex. The vertices are sorted in an octree that allows for fast spatial searching.

Given a region of influence, the octree is traversed to find all leaves that intersect the region. For every such a leaf we test the regions of all its vertices for intersection with the given region. Only the triangles that are incident to a vertex with an intersecting region must be considered for geometric intersection tests. This way, we can reject many triangles very early and very efficiently because we only need box-box intersection tests. When an edge collapses into a new vertex, the region of the new vertex can be easily computed by just adding two boxes.

4.2.1 Intersection-Free Triangle Mesh Simplification

The border of a tetrahedral mesh is a triangle mesh. We simplify the mesh with quadric error metrics by a sequence of edge collapses and embed the detection of self-intersections as follows.

Spatial Searching

We need a spatial search data structure that enables us to locate points and triangles quickly. Furthermore, it must support dynamic changes, i.e. simplices must be added and deleted from the data structure efficiently at run time because the simplification changes the triangle mesh and deletes and adds vertices or triangles.

For most tetrahedral meshes, the triangles at the border vary strongly in their sizes as shown in figure 4.6. We use an octree whose leaves adapt to the sizes of the triangles. We did not want to use a grid that could be implemented slightly easier but tends to oversample the coarse outer regions and undersample the highly detailled regions.

The octree contains just vertex indices. We do not need to store edges or triangles but can find them on-the-fly as will be explained later. This not only saves memory but also frees us from implementing specific routines that detect intersections between leaves and edges or triangles of the mesh. Just a simple point-in-box test is necessary.

We construct the octree by a run over the vertices of the triangle mesh and start with an octree that contains a single leaf which covers the slightly enlarged bounding box of the mesh. The box enlargement is necessary for later dynamic updates because new points may lie outside of the bounding box of the original mesh.

For each vertex, its octree leaf is detected and the vertex is inserted into this leaf. If the leaf contains more than a given number of vertices (which is typically fixed to 10), the leaf is regular subdivided at its mid-point into 8 sub-leaves.

For fast dynamic updates of the octree, each vertex of the triangle mesh stores the leaf index that it is assigned to. So, it can be deleted in constant time. If a vertex is re-inserted, the leaf of the vertex is located by a top-down traversal and the vertex is added to its leaf.

Intersection Detection

We assume that the border mesh is intersection-free because it has been part of a tetrahedral mesh before (otherwise we would have to remove these intersections first).

Each vertex of the triangle mesh has a region of influence assigned to it. This region must at least contain all triangles that are incident to this vertex. We chose axis-aligned bounding boxes to represent regions because they are easy to implement and provide extremely fast box-box intersection tests. To establish nomenclature, we say that the vertex v has its bounding box B_v as shown in figure 4.7a.

When an edge $e = (v_1, v_2)$ collapses, its two end vertices v_1 and v_2 collapse into a new point v_s. All triangles that are incident to v_1 and v_2 are deformed and the (at most) two triangles that are incident to both v_1 and v_2 are deleted. The deformed triangles may intersect other triangles which can be found quickly using the regions of influence.

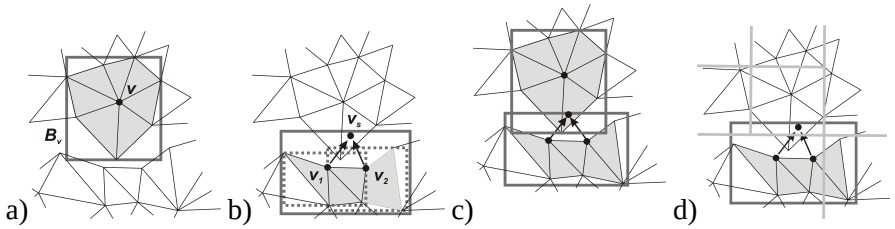


Figure 4.7: a) The region of influence contains the triangles that are incident to a vertex. b) If two vertices collapse, the region of influence of the new vertex is built by merging the regions of the collapsing vertices and adding the new point. c) The new region is tested for intersection with other regions. If an intersection occurs, at least one of these regions contains the intersecting triangle. d) Although the vertex is contained in exactly one octree leaf, its region may intersect several leaves.

The region of incluence of a new point v_s must be contained in the union of the regions of $v_{1,2}$ plus the new point v_s. Using bounding boxes, we merge the bounding boxes $B_{v_{1,2}}$ of $v_{1,2}$ into the new box B_{v_s} and add v_s to it. Given B_{v_s}, we use the octree to quickly reject all triangles that are definitely outside B_{v_s} and enumerate those triangles that are candidates for geometric intersection tests as follows.

The octree is traversed from top to down with B_{v_s}. Whenever the bounding box of a node or leaf intersects with B_{v_s}, the node is further traversed. Note that the bounding box of a node (or leaf) is defined as the sum of the bounding boxes of all vertices in its subsequent leaves (i.e. regions of influence) as will be described later. When a leaf is reached, we iterate over all its vertices and test their bounding boxes against B_{v_s}. Only if both boxes intersect, we enumerate the triangles around the vertex as candidates for geometric intersection tests.

This strategy eliminates very early a lot of triangles by simple box-box intersection tests. The enumeration of triangles around a vertex can be implemented efficiently with a half-edge data structure or something similar which is necessary for an edge collapse based simplification anyway.

The geometric intersection tests are taken from [GBK03] which reduces an triangle-triangle test to edge-triangle tests and point-inside-tetrahedron tests. If an intersection is detected, we first try

to avoid the intersection by moving the new point to a intersection-free position [GBK03] and, if this is not possible, we reject the edge collapse.

After the edge collapse, the vertices $v_{1,2}$ are removed from the octree and the vertex v_s is inserted into the octree. Its region of influence is left as B_{v_s} which is just an approximation to the bounding box which would tightly cover all triangles that are incident to v_s. But we found that using the approximation behaves well and saves the computational cost for recomputing the exact (tight) bounding box.

For the algorithm to work correctly, the bounding box of a node (or leaf) must contain all the bounding boxes of its vertices. Initially, these boxes are set to the union of the boxes of its vertices by a bottom-up traversal. At run time, the box of a node is simply enlarged by the box B_{v_s} attached to the new point v_s whenever B_{v_s} intersects the currently stored box.

4.2.2 Point Sampling

Given the simplified (and water-tight) boundary surface mesh, we want to point sample its interior such that the original tetrahedral mesh is well approximated by a constrained tetrahedralization of these points.

We want to find areas that can be represented by just one point who is a representant of all other points in this area such that the approximation of the tetrahedral mesh does not introduce an error that is bigger than some threshold. We distinguish between a field error and a space error, see section 3.1.1.

Typically, tetrahedral meshes have more than one attribute value per vertex. The burdigalian seaway has seven floating point attributes (four scalars and one vector), the F16 has six attributes (three scalars and one vector). If the simplification shall respect all attribute channels, the attributes need to be normalized to the unit interval $[0, 1]$. For scalar values, this is trivial. For vector values, we propose to find the largest vector, scale all vectors such that the largest vector has length 1 and shift the vectors to the unit interval by $\frac{1}{2}\mathbf{v} + 0.5$. We note the attribute values of a vertex as a single vector $\mathbf{a}$ that contains all attribute values (in any order).

We use an octree as spatial subdivision to find areas of similar attributes because it provides a regular subdivision that can adapt to local feature sizes and is thus often used for mesh generation.

Each leaf stores the minimal and maximal attribute values $\mathbf{a}_{min}$ and $\mathbf{a}_{max}$ together with all points that fall into the leaf. If the difference between $\mathbf{a}_{min}$ and $\mathbf{a}_{max}$ gets larger than a threshold, the leaf is subdivided regular at its midpoint and the vertices are spread over the eight children.

This produces a small field error but may leave a big space error especially in large cells as shown in figure 4.8. So, we want to estimate the space error. The first observation is that, given our sampling strategy (that is described soon), the extent of the box does not necessarily bound the space error. But the second observation is that the extent of the leaf relates to the space error. Points of small cells are located near to each other such that the space error tends to be small whereas points of larger cells are located far away from each other such that even small changes in the attribute values can cause large space errors.

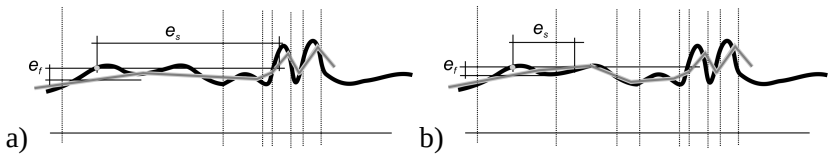


Figure 4.8: A 1D example: (a) the subdivision is created by a fixed threshold for the attribute error e_f. Although e_f is small, the space error e_s can become large especially in large cells. (b) The subdivision is adapted to decrease the space error.

So, we approximate the space error by a heuristic which scales the extent of the cell (relative to the total extent of the mesh) with the attribute difference. Although this is a rather crude heuristic to estimate the correct space error, it performs surprisingly well and produces a subdivision that also reflects thin structures in larger areas, see figure 4.8.

The field error e_f and the space error e_s steer the subdivision of an octree leaf:

$$\begin{aligned} e_f &= \|\mathbf{a}_{max} - \mathbf{a}_{min}\| \\ e_s &= D\,\|\mathbf{a}_{max} - \mathbf{a}_{min}\| \end{aligned}$$

where D is the relative extent of the cell where the space error e_s is calculated. The user provides for each error a threshold. A leaf is subdivided whenever one of the errors is bigger than its threshold.

All points of the original tetrahedral mesh that lie inside of the simplified border are inserted into the octree one after another. Note that the border surface may shrink during its simplification such that points of the tetrahedral mesh can lie outside of the simplified border (as it also has been noticed by [UBF+05]). In order to check if a point is inside the simplified border, we shoot a ray from the point into the direction that points away from the center of the tetrahedral mesh and count the number of intersections with border triangles. The point is inside of the simplified border if there is an uneven number of intersections, and outside otherwise.

After the octree has been constructed, its leaves contain interior points of the original mesh and each leaf contains points with similar attribute values. Now, we choose one point from each leaf that represents all the points of this leaf and is part of the tetrahedralization later.

For each leaf, we compute the average attribute value $\hat{\mathbf{a}}$ of all its points and take the point whose attribute values are closest to $\hat{\mathbf{a}}$.

4.2.3 Tetrahedralization

Given a set of sample points and a boundary triangle mesh, we perform a conforming Delaunay tetrahedralization that transforms this set into a tetrahedral mesh which respects the boundary triangle mesh. Instead of implementing an own tetrahedralizer, we use the TetGen library [Si05].

When the tetrahedralizer constrains the tetrahedral mesh to the faces of the simplified border mesh, it adds Steiner points to the mesh. Typically, these points lie near the border of the mesh

and the tetrahedralizer does not assign attribute values to them. A final run over the tetrahedra of the original tetrahedral mesh finds for each Steiner point its barycentric coordinates within its enclosing tetrahedron and uses these coordinates to compute the attributes.

4.2.4 Implementation

To save memory, we implemented the algorithm as a sequence of runs over the file that contains the tetrahedral mesh.

The first run reads and stores all vertices and constructs the vertex octree of section 4.2.2. Furthermore, we run over all tetrahedra and for each tetrahedron, its four faces are inserted into a hash table with their three vertex indices as hash key. If a face is detected that has already an entry in the hash table, this face is an internal face of the tetrahedral mesh (because the face-adjacent tetrahedron has added this face before) and can be deleted from the hash table. All faces that are stored in the hash table after the final tetrahedron has been read belong to the border triangle mesh. Because we have stored all vertices, we can now construct and simplify the triangle mesh.

The second run over the tetrahedra finds the attribute values of Steiner points. For each tetrahedron, we check if it has at least one vertex that belongs to the border. If it has such a border vertex, we check if one of the Steiner points is inside this tetrahedron. If we have left Steiner points after the second run, we finally iterate over all tetrahedra that have no border vertex and check if these tetrahedra contain a Steiner point.

4.2.5 Results

The simplifier has been tested with the six datasets of table 4.4. Both quality and performanced are reported. The timings are measured on a Pentium 4 (2.8 GHz) system with 1 GB main memory and Windows XP.

Quality.

The approximation quality of the simplified tetrahedral mesh is reported in table 4.4. For a discussion within this thesis, I implemented the approach of [UBF+05] as well as quadric error metrics for tetrahedral meshes and report the timings and average field errors. Note that our quadrics implementation is by far not the fastest because it runs rather expensive checks to avoid the creation of non-manifold or degenerated vertices and edges at the border of the tetrahedral mesh. For faster implementations we refer to [VCL+05, Pop03] which can reduce the Fighter for example within only 71 seconds to about 10% of its original size (our quadrics need 210 seconds). Nevertheless, the presented simplifier needs 12 seconds only (with intersection detection 32 seconds) and is thus at least twice as fast.

We approximate the mean field error (section 3.1.1) between two attribute fields as sum over

vertices

$$\tilde{E}_M^f = \sqrt{\frac{1}{|V|} \sum_{v \in M} \|\mathbf{a}(v) - \hat{\mathbf{a}}(v)\|^2}$$

where $|V|$ is the number of vertices in the original mesh M with attributes $\mathbf{a}$ while the simplified mesh has attributes $\hat{\mathbf{a}}$. All points that lie outside of the simplified mesh are projected onto the nearest point on the boundary. The approximation of the mean field error $\tilde{E}_M^f$ has also been used by [CCM+00, UBF+05].

Comparing to the results of Uesu et al. [UBF+05], we get similar results with respect to the mean field error of the scalar field. But for isosurface reconstruction, our space error ensures a much higher quality as reported in table 4.2 for the Fighter dataset (see also the colorplates). We computed nine isosurfaces and used the M.E.S.H. tool [ASCE02] to compute the mean geometric error d_M and one-sided Hausdorff distance d_O between the isosurfaces of the simplified mesh and the isosurfaces of the original mesh. Especially isovalue 0.2 is strikingly better reconstructed. The field error alone doesn't capture this isosurface which is embedded into an area of very similar attribute values.

Finally, we have a strict control over the domain error of the tetrahedral mesh which measures the geometric deviation of the boundary. If quadric error metrics are applied directly to tetrahedral meshes with additional face quadrics and vertex distribution quadrics as described in § 3.1.4, biasing artifacts may arise and the choice of the weight for these additional quadrics needs to be adjusted manually by the user. With my approach, the boundary is simplified independently from the interious which enables an easy control of the domain error.

Performance

The simplification timings are reported in tables 4.3 and 4.4. The simplification of the border took most of the time. We noted that the construction time of the tetrahedral mesh, i.e. the time of the conforming Delaunay tetrahedralizer, depends on the shape of the boundary surface mesh. The Sea dataset has a rather complex surface with many slivery triangles and sharp edges which forced the tetrahedralizer to slow down.

Uesu et al. [UBF+05] do not state if the points are incrementally inserted into the kd tree or if all points are first inserted into the root which is then recursively subdivided. I implemented both choices. The second choice gave better results which are listed in table 4.4 and are similar to mine. Nevertheless, the octree is incrementally constructed which results in the slightly better performance. I can simplify the Earthquake dataset with a peak memory usage of 300 MB which is much less of what has been reported by [UBF+05] who need 400 MB for the very much smaller Fighter dataset. My main part of memory usage is caused by the construction of the border surface which in reverse is caused by the bad mesh layout of the dataset.

The measured timings for the intersection detection correspond to the timings reported by [GBK03]. But my approach is simpler to implement because I only need to locate points within an octree and do not need to sort edges or triangles into cells. Additionally, my algorithm consumes (much) less

memory because the triangles that are candidates for intersection detection are enumerated at run time and do not have to be stored explicitly. The simplification pipeline slows down by a factor of 3 to 6 similar to [GBK03].

Isovalue	Octree		KD tree		Qem	
	d_M	d_O	d_M	d_O	d_M	d_O
0.1	0.33	3.1	0.79	3.2	0.29	1.55
0.2	***0.64***	***3.39***	***2.15***	***8.19***	***0.51***	***4.08***
0.3	0.23	1.34	0.32	1.95	0.43	1.7
0.4	0.14	0.88	0.15	1.12	0.15	1.05
0.5	0.14	0.81	0.09	0.45	0.12	0.77
0.6	0.24	1.18	0.26	1.05	0.25	1.22
0.7	0.08	0.35	0.18	0.84	0.04	0.25
0.8	0.06	0.21	0.14	0.52	0.04	0.12
0.9	1.69	6.69	1.78	7.3	2.36	9.66

Table 4.2: Nine isosurfaces of the Fighter dataset. The mean geometric error d_M and the one-sided Hausdorff distance d_O of an isosurface of the simplified mesh to the isosurface of the original mesh are reported. The errors are given relative to the bounding box of the original isosurface. Note how we can reconstruct isovalue 0.2 in contrast to [UBF+05]. The Fighter was simplified to 66K tets (octree) and 78K tets (KD tree) which is about 5% of its original size.

Name	#Border Faces Original	Time [sec] 1K ecol without intersection detection	Time [sec] 1K ecol with intersection detection
Sea	56,848	0.09	0.45
Post	16,796	0.11	0.41
Neghip	47,628	0.08	0.29
Fighter	83,504	0.10	0.31
F16	309,932	0.12	0.69
Earthquake	484,514	0.09	0.30

Table 4.3: The timings for the simplification of the border triangle mesh. Given in seconds for collapsing 1000 edges without and with intersection detection.

Name	# Vertices	# Tetra	Reduction to this percent	Octree			KD tree		Qem	
				$\tilde{E}_M^f$	Time [sec]	Time Percentage Border - Octree - Tetr	$\tilde{E}_M^f$	Time [sec]	$\tilde{E}_M^f$	Time [sec]
Sea	102,165	524,640	20%	0.21	19	57-01-40	-	border intersecton	0.09	34
Post	108,300	624,153	10%	0.15	15 (5)	84-01-15	0.14	(7)	0.10	98
Neghip	262,144	1,250,235	10%	0.03	18 (6)	85-01-14	0.031	(8)	0.021	175
Fighter	256,614	1,403,504	10%	0.013	32 (12)	78-01-21	0.012	(17)	0.007	210
F16	1,124,648	6,345,709	5%	0.08	150	86-01-13	-	border intersecton	0.05	923
F16	1,124,648	6,345,709	2%	0.11	157	87-01-12	-	border intersecton	-	-
Earthquake	2,461,694	13,980,162	1%	0.17	151 (49)	78-01-21	0.18	(62)	0.17	914 (§ 6.1.2)

Table 4.4: The mean field errors $\tilde{E}_M^f$ and timings of our octree approach, the KD tree [UBF+05], and quadric error metrics. A '-' means that we were not able to process the datasets. The percentage columns report the percentages of the steps with respect to the total run time. The timings in brackets are measured with border intersection tests switched off. Timings are given without file I/O (for the Earthquake dataset, reading the file and constructing the surface mesh takes about 15 seconds).

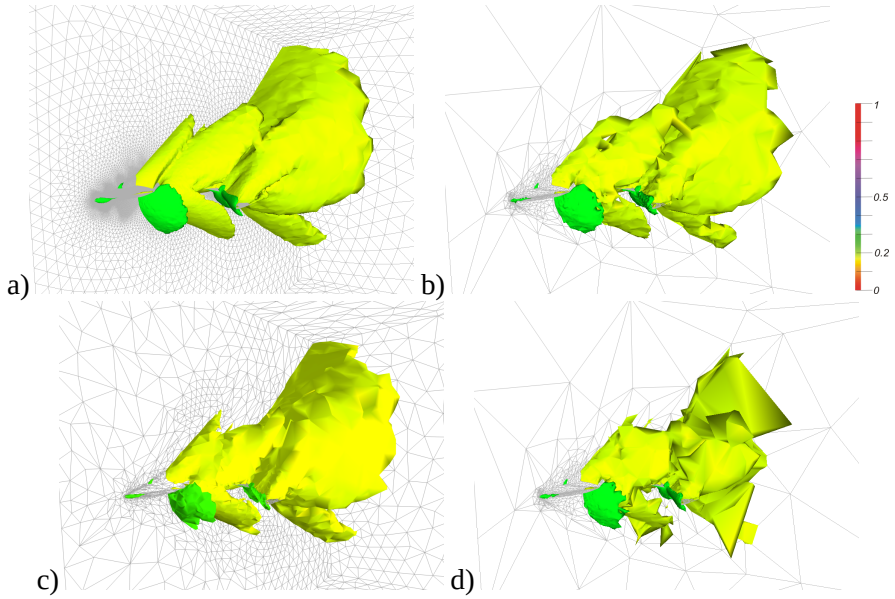
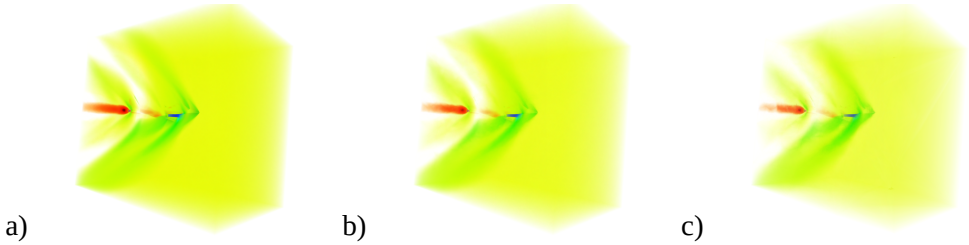
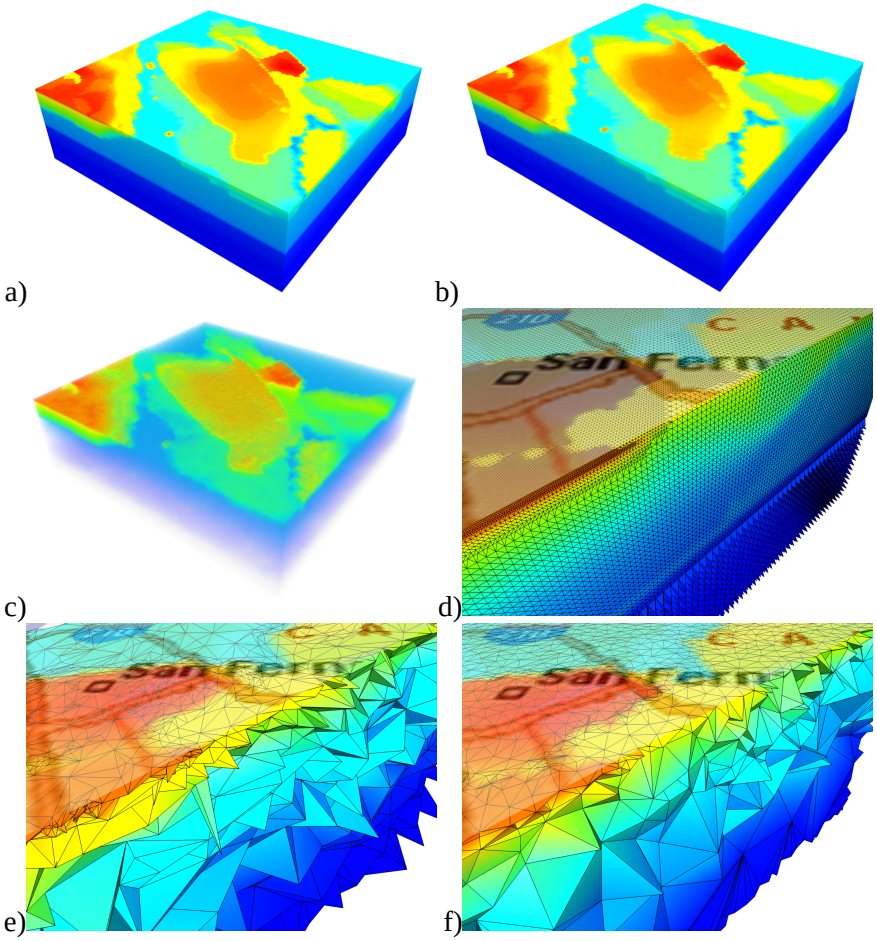


Figure 4.9: Two isosurfaces (at values 0.2 and 0.3) of the Fighter dataset are rendered. They are reconstructed from the original mesh (a), from our simplifier (b), from a quadrics-reduced mesh (c), and with a kd tree and field error only (d). Note the extreme distortion at the yellow isosurface in (d) which shows the reconstructed surface at isovalue 0.2.



a) b) c)

Figure 4.10: The direct volume renderings of the Fighter dataset show that the simplified mesh has only a small error in the scalar attributes. The image (a) is rendered with the full mesh, (b) with quadric error metrics (80K tets) and (c) with our approach (66K tets).



a) b) c) d) e) f)

Figure 4.11: The earthquake dataset contains 13 million tetrahedra. (a) shows the original mesh where the border triangular mesh is rendered with its colored attributes. The edges of the tetrahedra are not rendered because they are too dense. (b) shows the simplified mesh (surface-colored) while (c) shows a direct volume rendering of the simplified mesh (1%, in 49 sec). (d,e,f) show cuts of the dataset for the full mesh (d), a quadrics-simplified mesh (e) and a point-simplified mesh (f).

Chapter 5

Multi Resolution Models I

A good scientific theory should be explicable to a barmaid. - Ernest Rutherford

Beside pure simplification algorithms, multi-resolution models are a key tool to handle huge volume datasets and unstuctured tetrahedral meshes in particular, and allow visualization systems to be fully interactive during direct volume rendering or isosurface extraction. A multi-resolution model of such a mesh stores a coarse base mesh together with a collection of records describing how the base mesh can be refined by inserting new vertices and tetrahedra.

Using these records, a visualization system can refine or coarsen the visualized mesh to viewing or classification parameters. Different levels of detail connect to each other without gaps or overlappings and the model ensures that the adapted mesh is consistent at any time.

This chapter introduces two multi-resolution models for tetrahedral meshes. Both models are based on a progressive representation and work at vertex-level by using edge collapses and vertex splits as fundamental mesh update operations. Both models address a general problem of multi-resolution models: to encode the vertex splits compactly using a few bits only.

The first model described in section 5.2 – Predictive Tetra Mesh – introduces the concept of predictive vertex indices. Thereby, a vertex split predicts the vertices of all newly created tetrahedra from existing tetrahedra. It is shown that the prediction values can be compressed well.

The second model of section 5.3 – FastTetraMesh – extends the idea of FastMesh [Paj01] from pure triangle meshes to tetrahedral meshes. Restricting itself to full-edge update operations, a common data structure is used to store the mesh and allows all update operations to be calculated efficiently without the need to store any extra bits for split operations but for the cost of computing additional validity functions at run time.

Before describing both models in more detail, an overview to progressive representations of meshes and by extension multi-resolution representations is given as well as an introduction to dynamic meshing in general.

As extension of a classic simplification pipeline which simplifies a mesh M down to a coarse

approximation M_0 by a sequence of edge collapses $ecol_i$

$$M = M_n \overset{ecol_{n-1}}{\longrightarrow} M_{n-1} \overset{ecol_{n-2}}{\longrightarrow} ... \overset{ecol_0}{\longrightarrow} M_0,$$

a major contribution of Progressive Meshes [Hop96] is the notation of a mesh as a sequence of $vsplit_i$ records that can be applied to the coarse base mesh M_0 in order to recreate the full resolution mesh M:

$$M_0 \overset{vsplit_0}{\longrightarrow} M_1 \overset{vsplit_1}{\longrightarrow} ... \overset{vsplit_{n-1}}{\longrightarrow} M_n = M.$$

Such a representation is called *progressive representation*. Each edge collapse stores all update information in a $vsplit_i$ record which is necessary to undo the collapse and to refine the mesh again by applying the vertex split $vsplit_i$. Multi-resolution models like [Hop97, ESV99, KL01, Paj01, DDF02, DDFM$^+$05] can be considered as an extension of progressive meshes where $vsplit_i$ can be applied in an order different from the strict progressive representation. In addition, $ecol_i$ can be applied again to coarsen the mesh. The information recorded with each update must be sufficient to

1. Rapidly perform edge collapses and their inverse operations, vertex splits, on a currently adapted mesh in order to enable a visualization system to remain interactive.

2. Ensure the topological and geometrical validity of all operations. This typically includes a fast inspection of all tetrahedra incident to the collapsed edge or split vertex. Otherwise, manifold meshes can become non-manifold or geometric distortions like element flips can occur.

3. The adapted mesh is the nearly minimal mesh that fulfills the given error criterion based on the viewing parameters in order to achieve an optimal display rate during visualization.

A visualization system for direct volume rendering or isosurface extraction uses the multi-resolution model to achieve interactivity during rendering while the visible error is below a user-specified threshold. Instead of rendering the mesh at its full resolution, the mesh is coarsened and refined by applying edge collapses and vertex splits until the adapted mesh can be visualized with an error acceptable for the user (or no error at all). A mesh with a minimal number of tetrahedra fulfilling the error criteria is called a *minimal mesh*.

Using edge collapses and vertex splits allows for a very fine-grained control over the adaptation process. Given classification and viewing parameters, the visualization system can adapt the mesh to a nearly minimal mesh. Nevertheless, if the adaptation takes place on the CPU, the complete adapted mesh still needs to be transferred onto the GPU for rendering.

5.1 Basics on Dynamic Meshing

Basically, the collapse of an edge $\{p, q\}$ merges both extreme vertices p and q into a new vertex v. We denote an edge collapse as $ecol(p, q, v)$ meaning that p and q collapse into v. Its inverse

operation vertex split recreates the edge and is denoted as $vsplit(v, p, q)$. Here, the vertex v is called split vertex.

There are two different kinds of edge collapses. A *full-edge collapse* can choose the position of the new vertex v freely and has always $p, q \neq v$. In contrast, an edge can be split into two half-edges of inverse orientation, i.e. into half-edges (p, q) and (q, p). A *half-edge collapse* collapses a half-edge (p, q) into p (whereas the half-edge (q, p) would collapse into q). Using the notation above, a half-edge collapse can be written as $ecol(p, q, p)$.

All tetrahedra that are incident to the edge $e = \{p, q\}$ and thus to both extreme vertices p and q are called *fan tetrahedra* of e. A tetrahedron is a *modified tetrahedron* if an edge collapse replaces exactly one of its vertices by another vertex. For a full-edge collapse, all tetrahedra incident to exactly one of the extreme points are modified tetrahedra while in the case of a half-edge collapse (p, q), only the tetrahedra incident to q are modified (tetrahedra that are incident to p remain unchanged).

From a meshing point of view, the collapse of an edge $\{p, q\}$ changes the mesh with

1. Remove both extreme vertices p and q of the collapsed edge from the mesh.
2. Introduce a new single vertex v that the edge collapses into.
3. Replace the extreme vertices p and q within all tetrahedra incident to p and q with the new vertex v. Tetrahedra that are incident to both p and q degenerate to triangles and can be removed. Update the adjacency information between any pair of tetrahedra that are connected by such a triangle.

A vertex split inverts an edge collapse and changes the mesh

1. Delete the split vertex v.
2. Introduce both extreme vertices p and q of the new edge.
3. Out of all tetrahedra incident to v, identify the set T_p of tetrahedra where v must be replaced by p and the set T_q of tetrahedra where v must be replaced by q. T_p (resp. T_q) is the set of tetrahedra that have been incident to p (resp. q) only at the time of the corresponding edge collapse.
4. For all tetrahedra in T_p (and T_q), replace the vertex v with p (and q, respectively).
5. Create all tetrahedra that have been deleted, i.e. create all fan tetrahedra of the edge, and set their vertex indices.

Because not every edge collapse is valid, the geometric and topological preconditions of § 3.1.3 must be ensured.

The various approaches presented so far differ mainly in how split items 3 – 5 are dealt with, i.e. how a vertex split is calculated correctly. Finding both sets T_p and T_q is by no means a trivial task

because at the time of a vertex split, all tetrahedra incident to the split vertex v reference only v and cannot be distinguished from each other. Additional information needs to be stored with each update record that enables both sets to be identified. Furthermore, inserting the fan tetrahedra and updating all their adjacencies (between fan tetrahedra themselves as well as to tetrahedra of T_p and T_q) is neither a trivial task.

The situation for tetrahedral meshes differs from the split situation for triangular meshes in the non-fixed number of fan tetrahedra. In triangular meshes, there are at most two fan triangles whereas there are more than two (in average 6) fan tetrahedra. So, specialized techniques for triangular meshes relying on surface orientations cannot be transferred to tetrahedral meshes directly.

All subsequent sections describe algorithms for unstructured datasets and can be applied to more general tetrahedral meshes than algorithms that focus on tetrahedralizations of regular (voxel–based) datasets like [MDM04, ZCK97].

5.2 Predictive Tetra Mesh

As contribution to well-known multi-resolution models at vertex level, we exploit the redundancy present in the connectivity information of a mesh in order to split a vertex. So, a vertex split can be computed quickly by just comparing vertices which reduces computational time. In contrast to [CMRS03], we only need to sort fan tetrahedra and do *not* need to walk over tetrahedra in any particular order.

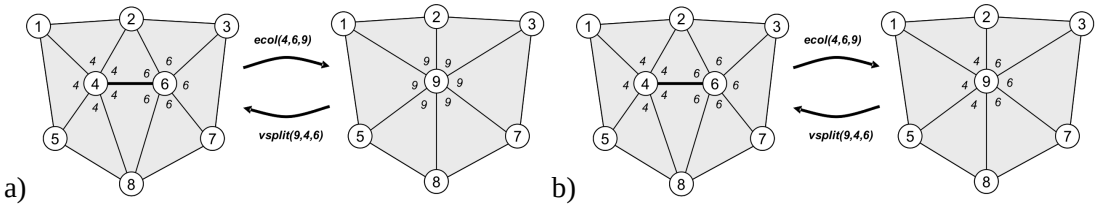


Figure 5.1: (a) A classic edge collapse overwrites the indices. (b) If the indices are left unchanged, they immediately reveal the correct partition of the tetrahedra for a $vsplit$.

Figure 5.1 summarizes the basic idea. For clearness of visualization, the example shows a triangular mesh. Using a standard indexed data structure for tetrahedral meshes (like [Nie97]), the index of a vertex occurs in all tetrahedra that are incident to it. For example, the index 4 is stored five times as shown in figure 5.1a. If an edge collapse like $ecol(4, 6, 9)$ overwrites the stored indices, the data structure does not reveal how the tetrahedra are to be partitioned into the sets T_6 and T_4 for the inverse operation $vsplit(9, 4, 6)$. The information about this partition has to be stored explicitly. In contrast, our algorithm does *not* overwrite stored indices during $ecol(4, 6, 9)$ as visualized in figure 5.1b and exploits the redundancy of the indexed data structure to split the vertex 9. The information about the partition of the split tetrahedra around vertex 9 into the sets

T_4 and T_6 can be queried directly from the indexed data structure. All tetrahedra with a 4 belong to T_4 while all tetrahedra with a 6 belong to T_6.

5.2.1 Construction

A progressive simplification reduces the mesh complexity by a sequence of edge collapses with quadric error metrics. An edge collapse may be invalid due to geometrical and topological restrictions as described in § 3.1.3. If a multi-resolution model is constructed by a progressive simplification, additional care must be taken in order to introduce as few dependencies between updates as possible. Because these dependencies mainly depend on mesh elements in the link of a collapsed edge [XV96], the concept of independent edge collapses as introduced in § 4.1.1 can be applied again with a slightly widened definition of the term independent edge.

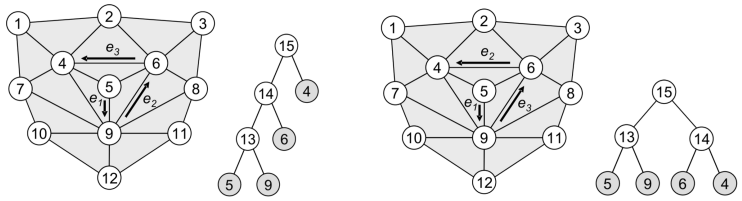


Figure 5.2: Left: a deep hierarchy with many direct dependencies, right: a more balanced hierarchy as created by independent edge collapses of § 4.1.1.

Independent Edges

Given two edges $e_1 = \{p_1, q_1\}$ and $e_2 = \{p_2, q_2\}$ together with the unions $S(e_i)$ of the stars of their extreme vertices $S(e_1) = St(p_1) \cup St(q_1)$ and $S(e_2) = St(p_2) \cup St(q_2)$, the edges e_1 and e_2 are *independent* iff the cut of both unions is empty, i.e. $S(e_1) \cap S(e_2) = \oslash$ as illustrated in figure 5.3.

Collapsing independent edges separates their region of influence, that is, they modify different mesh elements. Remember that an update depends on another update if their regions of influence overlap, see § 3.2. Independent edges decrease the number of dependencies between updates and keep the depth of the binary hierarchy flat. It is advantegous to identify dependent edges of a collapsed edge instead of independent edges. An edge depends on a collapsed edge $\{p, q\}$ iff it is incident to any vertex $v \in L(e) = Lk(p) \cup Lk(q)$ as shown in figure 5.3c.

The simplifier of § 4.1.1 can be extended to support collapses for this type of independent edges. Especially in regions with degenerated mesh parts like flat boundaries, the depth of the vertex hierarchy is decreased by independent collapses.

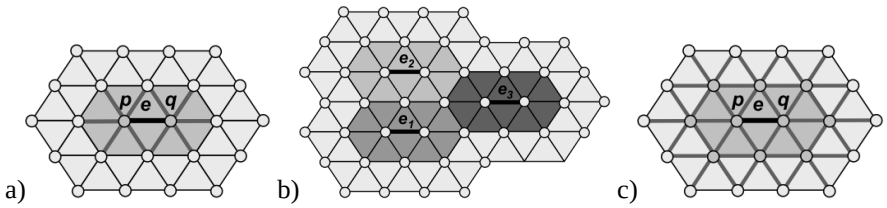


Figure 5.3: (a) $S(e) = St(p) \cup St(q)$. (b) The edges e_1, e_2 and e_3 are independent because $S(e_1)$, $S(e_2)$ and $S(e_3)$ do not overlap. (c) An edge depends on e iff it is incident to a vertex of the union of the links $L(e) = Lk(p) \cup Lk(q)$. Dark vertices belong to $L(e)$ while dependent edges are drawn thick gray.

Binary Vertex Hierarchy

During progressive simplification, a binary vertex hierarchy is recorded consisting of an update node for each vertex. An update node and its vertex have the same index and can be identified by this index. The hierarchy is constructed by adding the nodes of both extreme vertices p and q of an edge collapse $ecol(p, q, v)$ as children to the node of the new vertex v as shown in figure 5.4. The leaf nodes correspond to the original vertices of the mesh whereas interiour nodes correspond to vertices creates by edge collapses.

The vertices of the input mesh can be enumerated in any order whereas new vertices are enumerated ascendingly in the order given by the sequence of edge collapses (compare [ESV99, CMRS03]).

5.2.2 Dynamic Meshing

Each node of the hierarchy carries update information such that $ecol$ as well as $vsplit$ can update the mesh as follows.

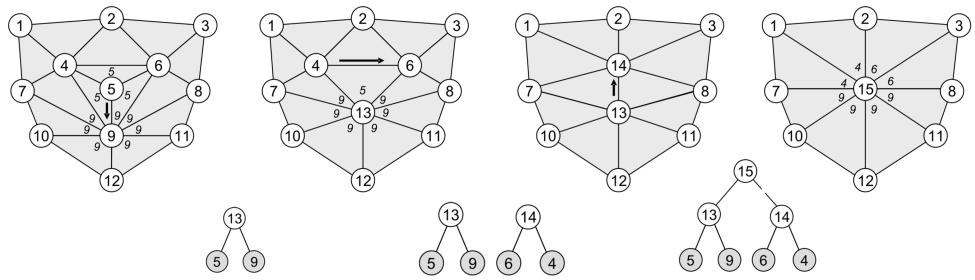


Figure 5.4: The mesh is simplified by a sequence of edge collapses. Each edge collapse adds a new update node to the binary vertex hierarchy. The index of each vertex is stored in each tetrahedron (denoted as triangles in the pictures) that is incident to it. An $ecol$ does not overwrite the stored indices but leaves them unchanged.

For a $vsplit(v, p, q)$, we need to partition the tetrahedra that are incident to v into the sets T_p

and T_q. Remember that the set T_p contains all tetrahedra that have been only incident to p while the set T_q contains all tetrahedra that have been only incident to q at the time of the corresponding $ecol(p, q, v)$. Instead of storing the partition information explicitly with each update, I exploit the hierarchy and the redundancy of the tetrahedral data structure to find this partition of tetrahedra.

As a key part of predictive tetra meshes, an $ecol(p, q, v)$ does not overwrite the vertex indices in affected tetrahedra as shown in figure 5.4. Figure 5.5 shows a particular situation where $ecol(4, 6, 14)$ leaves the vertex indices 4 and 6 unchanged in all affected tetrahedra.

Consider that vertex 14 is to be split and that all preconditions are fulfilled, i.e. the indices of incident vertices are lower than 14. The tetrahedra that are incident to vertex 14 have currently 4 and 6 stored. The correct partition splits the tetrahedra right between the tetrahedra with vertices 4 and 6 such that the tetrahedra with vertex 6 belong to T_6 while the tetrahedra with vertex 4 belong to T_4.

Looking at the vertex hierarchy in figure 5.5, we see that vertex 4 belongs to the right and vertex 6 belongs to the left subtree of split vertex 14. The vertices of the right subtree belong to one set while the vertices of the left subtree belong to the other set.

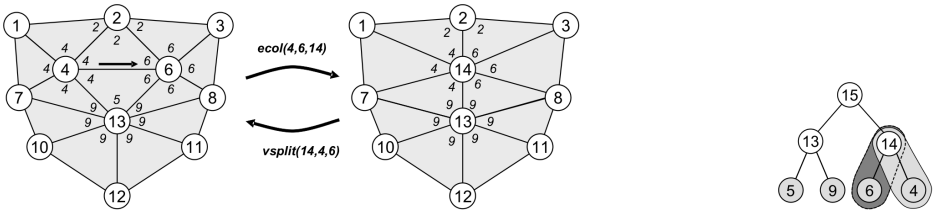


Figure 5.5: The situation for $ecol(4, 6, 14)$. The sets T_4 and T_6 can be found immediately from the stored indices. However, the last tetrahedron that stores index 5 is removed.

This observation can be generalized. The partition of the tetrahedra of a $vsplit(v, p, q)$ can be found by looking at the leaves of the left and the right subtree of v. The left subtree was formed by p while the right subtree was formed by q as described above. Thus, if an incident tetrahedron has a vertex that is a leaf of the left subtree of the vertex to be split, it belongs to the set T_p, and if an incident tetrahedron has a vertex that is a leaf of the right subtree of v, it belongs to the set T_q. For instance, the $vsplit(15, 13, 14)$ in figure 5.4 can find the partition immediately as $T_{15} = T_{\{4,6\}}$ and $T_{15} = T_{\{9\}}$.

Given the partition into the sets T_p and T_q, a fan tetrahedron is created inbetween any two face-adjacent tetrahedra that belong to different sets. This can be accomplished by a walk which crosses all faces. In principal, this walk does not have to have any special ordering.

Finally, a $vsplit$ has to choose the indices of all fan tetrahedra. Because the indices must be equal to the indices stored before the corresponding $ecol$ has been applied, the indices of a fan tetrahedron could be overtaken from the indices of its two adjacent tetrahedra in T_p and T_q. There is a final problem with this.

An $ecol$ can delete indices of a fan tetrahedron, i.e. an $ecol$ erases the last tetrahedron that has a

particular index stored. For instance, $ecol(4, 6, 14)$ deletes the index 5 in one of its incident tetrahedra as shown in figure 5.5. Because an $ecol$ can erase indices, the indices of a fan tetrahedron cannot be simply overtaken from its adjacent tetrahedra but must be predicted from them. This situation is illustrated in figure 5.6.

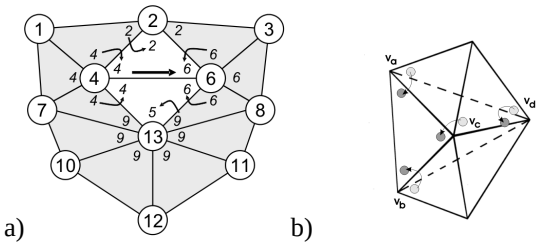


Figure 5.6: (a) The indices of the fan tetrahedra are to be predicted from the incident tetrahedra. Situation for $vsplit(14, 4, 6)$. (b) For a tetrahedron, three indices are predicted from the incident tetrahedron $(v_a, v_c, v_d, .)$ and one index from $(v_b, v_c, v_d, .)$.

The differences between the indices of a fan tetrahedron and its neighboring tetrahedra are recorded during preprocessing. Because these differences tend to be small, they can be compressed well. Predicting the indices of a single fan tetrahedron needs four prediction values which are usually integer values with a length of four bytes each. The four bytes are considered to be four characters which can be compressed well using arithmetic coding. The table of probabilites has 256 entries and stores for each possible 8-bit character value its probability. Each update stores the arithmetic compressed prediction values of all fan tetrahedra that need to be created.

Because the fan tetrahedra don't have a canonical ordering, we must define an order that is used to store the differences within an update. Each fan tetrahedron has two vertices v_c and v_d that are not $p(= v_a)$ or $q(= v_b)$, see figure 5.6b. v_c and v_d are interpreted as an ordered pair (v_c, v_d) if $v_c < v_d$ or (v_d, v_c) if $v_d < v_c$. These pairs are sorted lexicographically and thus define a unique order of the fan tetrahedra. We experienced that the average number of fan tetrahedra is 6 so that this sorting overhead is small (for instance, [CMRS03] needs to order all incident tetrahedra of a split vertex v lexicographically and restricts the number of incident tetrahedra to 64).

Evaluated by experiments, in average one quarter until one sixth of all updates do not need to store any prediction values because the indices of their fan tetrahedra and the corresponding adjacent tetrahedra are the same, interestingly mostly independent of the size of the mesh. This is exploited by the implementation which doesn't need to store any encoded differences for these updates.

Note that the prediction values depend on the distribution of vertex indices within a mesh. If similiar indices are located close to each other, the prediction values tend to be small, whereas if similiar vertex indices are far away from each other, the vertices are likely not be collapsed and thus the prediction values need larger and worse distributed values.

5.2.3 View–dependent Meshing

A valid mesh corresponds to a vertex front through the hierarchy as shown in figure 5.7.

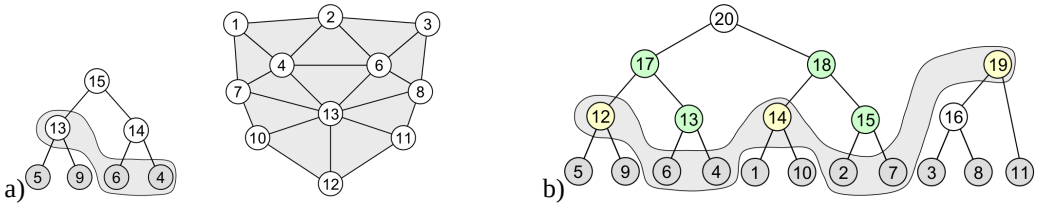


Figure 5.7: The vertex front through a binary vertex hierarchy corresponds to a mesh.

A vertex is said to be active if it belongs to the vertex front. The vertex front contains for every path from a root of the hierarchy to the leaves exactly one active vertex and for the path from a leaf to its root also exactly one active vertex. The vertices on the vertex front correspond exactly to the vertices in the current mesh, i.e. the mesh contains only these vertices while vertices that are not on the front are also not in the mesh.

Initially, the vertex front contains all roots of the hierarchy which correspond to the base. An adapted mesh can be created by moving the vertex front up and down by edge collapses and vertex splits, respectively. The validity of $ecol$s or $vsplit$s can be ensured by comparing vertex indices allowing the topological and geometrical preconditions of § 5.1 to be checked implicitly [XV96]:

- A $vsplit(v, p, q)$ is valid iff v belongs to the current mesh and the indices of all vertices v_i that are adjacent to v are lower than v, i.e. $v_i < v$.

- An $ecol(p, q, v)$ is valid iff p and q belong to the current mesh M_i and an edge $e = \{p, q\} \in M_i$. For each vertex v' that is adjacent to p or q, either v' is a root of the hierarchy or the index of the parent of v' is greater than the index of v.

A $vsplit(v, p, q)$ can be performed by forcing the $vsplit(v_i, p_i, q_i)$ operations of all vertices v_i that are adjacent to v and $v_i > v$. An $ecol(p, q, v)$ cannot be forced in a similar way but can only be applied if its preconditions are fulfilled.

Updates on the front are candidates for splits (yellow updates in figure 5.7) while updates at the hierarchy level above the front are candidates for collapses (green updates in figure 5.7).

The selection of $ecol$s and $vsplit$s depends on run–time factors like viewing parameters or classification parameters. A cost is assigned to each update that depends on the view–frustum, a region of influence and the field error of the update. Every update stores a radius of influence, that is, the radius of the minimal sphere that contains all mesh elements of the link of its split vertex, and a field error, i.e. the error of the attribute field that the edge collapse introduced. The calculation of this field error for an $ecol(p, q, v)$ measures the field values of all leaf vertices of the subtree with root v in the approximating mesh. The field error $E_f(p)$ for a single such leaf vertex p is the absolute difference between its approximation and its original field value. The field

error $E(v)$ of the update v is then simply the maximal value of all these leaf vertex field errors, i.e. $E(v) = \max_{p \in Leaves(v)}\{E_f(p)\}$.

Basically, we can adapt the mesh for direct volume renderings (DVR) or isosurface extractions. In order to define a cost per update in a DVR setting, the field error $E(v)$ is weighted with the current opacity classification value $C_\alpha(\Phi(v))$ (where $\Phi(v)$ is the field value of v) resulting in a weighted field error τ_f for the active vertex v

$$\tau_f = E(v) * C_\alpha(\Phi(v)).$$

An update with a completely transparent field value has always an error 0 and can be coarsened while an update with an opaque field value has its full error and might be refined. If the bounding sphere is completely outside the view-frustum, τ_f is set to 0 for coarsening. In order to decide which update is to be refined or coarsened, a hysteresis scheme is applied. If τ_f exceeds a user–specified threshold T_1, the vertex is refined, and if τ_f falls below another user–specified threshold $T_2 < T_1$, the vertex is coarsened. Otherwise, the vertex remains unchanged.

For isosurface extraction, the classification function $C_\alpha(\Phi(v))$ can be set to an isosurface classification function with $C_\alpha(i) = 1$ for an isovalue i and $C_\alpha(j \neq i) = 0$ for any other attribute values j.

A refinement queue keeps track of all updates that are candidates for splits sorted by their costs from highest to lowest. A coarsening queue stores all updates that are candidates for collapses sorted by their costs from lowest to highest. The adaptation of the mesh first collapses as many updates from the coarsening queue before it splits as many updates as possible from the refinement queue. Whenever an update is coarsened or refined, the queues are updates to correspond to the new vertex front. Note that especially vertex splits may force other vertices to split which results in several changes of the vertex front and queues.

Because we never overwrite stored indices, the current active vertex number needs to be computed efficiently for a stored index, i.e. the vertex that is currently active in the vertex front from a leaf upwards to its root. Therefore, every leaf of the vertex hierarchy has its current active ancestor stored. An $ecol$ or $vsplit$ operation updates this information by storing the new active ancestor in all leaves that are indexed in the current sets T_p and T_q. Note that not all leaves of the left or right subtree of an actual considered vertex need to be updated but only those that are currently referenced by tetrahedra. All other leaves are updated by succeeding operations when it is necessary to update them.

The described techniques can represent half-edge collapses as well as full-edge collapses. Only the information that is to be stored in the updates differs as shown in the following section.

5.2.4 Implementation Details

To get more concrete, a principal data structure for the updates is shown in figure 5.8.

A class hierarchy defines a unique interface to all updates such that they can be stored in a single list. Every subclass of `Update` only stores the needed information. So an update entry of a vertex

```
class Update; // the base class enables to store all updates
              // in a single vertex front list
class OrigUpdate :  Update { // for original vertices only
 int active;                 // index of the active vertex
 int parent;                 // parent update
};
class ZeroUpdate :  Update { // predicts always zero
 float error;                // field error
 float radius;               // radius of influence
 vector3 diff[2];            // pos_child = pos_parent + diff
 float attrdiff[2];          // attr_child = attr_parent + attrdiff
 int deep, wide;             // tree connectivity
};
class FullUpdate :  ZeroUpdate { // update predicts non-zero values
 char partition[6];          // encoded prediction values
};
```

Figure 5.8: The principal structures for updates.

of the original mesh need only to store the index of parent and the radius of the bounding sphere that contains all incident tetrahedra. But even this radius can be set to zero.

The updates of split vertices that always predict zero values can be encoded by `ZeroUpdate`, i.e. the indices of the incident tetrahedra do not differ from the indices of the fan tetrahedra. The other updates need to store the encoded sequence of predictions.

The binary vertex hierarchy itself can be implemented as an array of updates where the tree information is stored in the `deep` and `wide` entries. The index `deep` points always to the left child while `wide` points to the right sibling if the update is a left child or to the parent if the update is a right child [DDF02].

The algorithm starts with the base mesh that has the tetrahedral mesh stored as it has been left by the simplification, i.e. the indices of the vertices of the tetrahedra in the mesh have never changed (and are just represented by their active vertices in the vertex front) and can be used to split vertices. The vertex front is initialized to contain all root updates.

5.2.5 Results

The technique was tested with five datasets of different sizes each. The simplification timings were measured on a Pentium 4 with 2.8 GHz and 1 GB memory.

At runtime an edge collapse or a vertex split takes about 0.015 ms which corresponds to about 66K edge collapses / vertex splits per second. If the average number of tetrahedra incident to an edge is assumed to be 6 (which is the case for all of our models), Predictive Tetra Mesh can delete

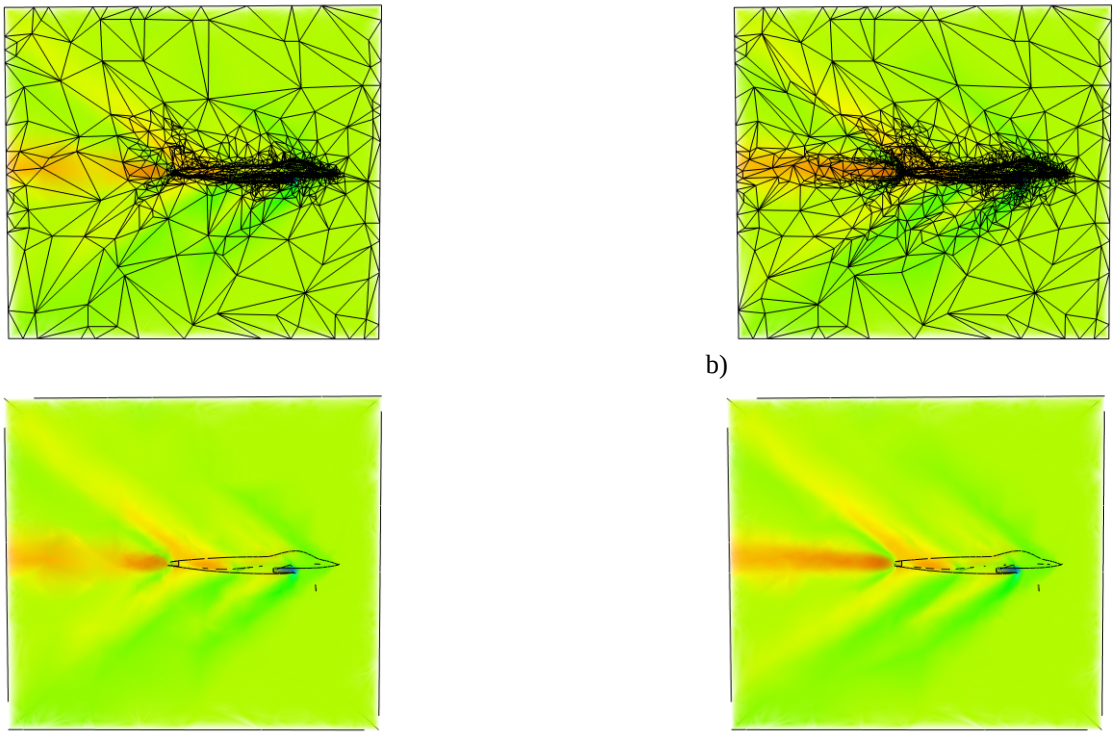


Figure 5.9: The base mesh contains a large field error (a, 30K tetrahedra) which is decreased by an adapted mesh (b, 80K tetrahedra). (c) and (d) show the corresponding direct volume renderings.

and / or add up to 400.000 tetrahedra per second. If the viewing parameters change slightly, the overhead for adapting the mesh is small because it a small amount of tetrahedra must be updated only.

In terms of memory consumption, a vertex is stored with three floating point values and a tetrahedron with four indices plus four indices for adjacent tetrahedra. Our data structures need 12 bytes to store the coordinates of a vertex and 32 bytes to store the indices and adjacencies of a tetrahedron. A OrigUpdate needs 8 bytes, a ZeroUpdate needs 48 bytes, and a FullUpdate additional 6 bytes to encode the differences as reported in table 5.1.

Comparing to [CMRS03] the only value of interest for comparision is the amount of memory that is needed to find the partition of tetrahedra around a split vertex because in principle [CMRS03] use similar data structures (they also store the field error, the differences of the vertex positions and atribute values, and the tree structure).

To store the partition information, [CMRS03] use a bit field of 64 bits (8 bytes). Every tetrahedron incident to the split vertex uses one bit that indicates which set the tetrahedron belongs to. The number of tetrahedra that are incident to a vertex is limited by 64. The tetrahedra must be sorted in order to to establish an order used to map each tetrahedron to its bit. Therefore, the tetra-

Figure 5.10: The mesh is adapted when the observer moves.

hedra are sorted lexicographically by their vertices where the vertices of a tetrahedron themself are ordered ascending.

Because my data structure needs at most 6 bytes (48 bits) currently, it improves upon [CMRS03] by a percentage of 25%. But remember that the F16 could be packed into at most 5 bytes and the Bucky ball into even fewer 4 bytes which is half the storage cost of [CMRS03] (concerning partition information).

In addition, Predictive Tetra Mesh does not need to sort all tetrahedra incident to a split vertex lexicographically but needs to sort only a small subset of them, in average 6 tetrahedra in contrast to 22 tetrahedra of [CMRS03]. (The average values are taken from table 2.3.) Note furthermore that about one sixth of all updates don't need to store anything and hence don't need a sorting at all.

Name	Bucky Ball	Blunt Fin	Sea	Fighter	F16
# Vertices	32,767	40,960	102,165	256,614	1,124,648
# Tetra	176,856	187,395	524,640	1,403,504	6,345,709
MHD	13	14	19	17	61
# Tetra Base Mesh	10K	14K	20K	50K	100K
# Updates	60K	80K	200K	500K	2,000K
Preproc. Time	47 sec	48 sec	156 sec	359 sec	1,014
# Max Bits	28	19	46	32	37
# Avg Bits	10.32	8.52	15.7	10.7	12.4
# Zeros	10K (16%)	20K (25%)	26K (13%)	80K (16%)	341K (17%)
File Size [MB]	3.7	4.0	13.2	27.8	140.1
File Size Orig [MB]	3.2	3.5	12.1	25.9	134.2
Time Base→Orig [sec]	0.95	1.09	3.05	7.7	31.4

Table 5.1: Properties of five sample datasets. MHD is the maximal hierarchy depth. # Updates is the number of hierarchy nodes without leaves. Max Bits reports the maximal length of an arithmetic encoded prediction array (out of all update nodes) while Avg Bits reports the average number of bits needed to encode the prediction array. The Zeros row lists the number of updates that don't need a prediction because the indices of all fan tetrahedra are equal to the indices of their predictor tetrahedra.

5.3 FastTetraMesh

FastTetraMesh stores the full tetrahedral mesh in an indexed data structure with adjacencies. This data structure is updated by edge collapses and vertex splits in order to coarsen or refine the mesh but no mesh elements are deleted physically from main memory. All mesh elements remain stored but can be deactivated by updates. Because all mesh elements remain stored, they can be queried quickly in order to perform vertex splits rapidly. This way, FastTetraMesh is an extension of FastMesh for triangular meshes [Paj01].

In order to ensure correctness of all update operations, FastTetraMesh is restricted to half-edge collapses. In a preprocessing step, a binary hierarchy of half-edge collapses is constructed which contains half as many update nodes than a vertex hierarchy.

In addition, edges are allowed to collapse whenever the edge fulfills all topological and geometrical conditions. This enables edges to collapse whenever these conditions are met. Other algorithms often prohibit edge collapses if the current neighborhood does not equal the neighborhood at simplification time, see [CMRS03, XV96] and § 5.2.

5.3.1 Dynamic Meshing

An indexed data structure with adjacencies [Nie97] stores for each tetrahedron its four vertex indices and its four adjacent tetrahedra. Using 4-byte integers as indices, the data structure needs $32n$ bytes in total if n is the number of tetrahedra. Figure 5.11 shows a part of such a data structure

covering three tetrahedra.

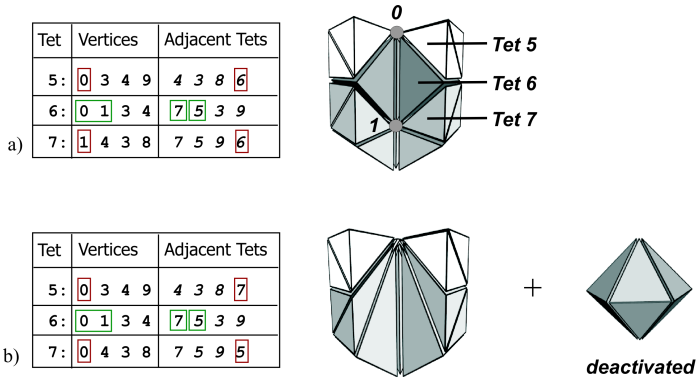


Figure 5.11: (a) Before $ecol(0,1,0)$: The fan tetrahedron 6 stores face-adjacent tetrahedra 5 and 7. (b) After $ecol(0,1,0)$: The fan tetrahedron 6 still stores face-adjacent tetrahedra 5 and 7 while those reference each other.

The data structure stores the four vertex indices of a tetrahedron in any order (but with an orientation that ensures a positive volume), for instance $t = (p, q, r, s)$. Each tetrahedron is adjacent to (at most) four tetrahedra where each adjacent tetrahedron is adjacent to exactly three of the four vertices $\{p, q, r, s\}$. We denote an adjacent tetrahedron that is incident to $\{p, q, r\}$ as $t = o(s)$. The indices of the four adjacent tetrahedra are stored in such an order that the index of an adjacent tetrahedron is placed at the position of the vertex index that is not incident to this tetrahedron, i.e. $(o(p), o(q), o(r), o(s))$.

A half-edge h is a directed edge $h = e = (p, q)$. All tetrahedra that are incident to h are called fan tetrahedra, as before. Beside the identification of a half-edge by its two extreme vertices p and q, a half edge can also be identified by the index of a fan tetrahedron and a local index that specifies the half-edge within the fan tetrahedron, that is, the local index selects the half-edge pair (p, q) out of all possible 12 half-edge pairs within a tetrahedron. Formally, a half-edge $h = (p, q)$ can be identified by $h = (t, l)$ with the tetrahedron index t and the local index $1 \leq l \leq 12$.

Given a half-edge $h = (t, l)$, the collapse of this half-edge identifies the extreme vertices p and q and replaces the vertex q with p in all tetrahedra that are incident to q. Thereby, all tetrahedra that are incident to h are flattened and degenerate to triangles. The two tetrahedra that are connected to such a flattened triangle are connected by setting their adjacency information. Figure 5.11 illustrates such an collapse and the updates of the data structure for a single half-edge collapse. Note that a half-edge collapse in FastTetraMesh does not overwrite the indices of fan tetrahedra.

As a key concept of FastTetraMesh, tetrahedra that are incident to h are not deleted from memory by a collapse but are marked as deactivated. They – and especially their adjacency information – remains stored in the data structure and enables a fast vertex split as follows. Given again a half-edge $h(t, l)$ where t is the index of a fan tetrahedron that has been deactivated by the corresponding half-edge collapse, the extreme vertices p and q can be identified within $t = \{p, q, r, s\}$. The local

index l allows to select two adjacent fan tetrahedra $o(r)$ and $o(s)$ of t that are opposite to r and s. All fan tetrahedra can be traversed starting at the tetrahedron t. Each fan tetrahedron still stores two adjacent tetrahedra $o(p)$ and $o(q)$ that are opposite to p and q, respectively. The adjacency information of these two adjacent tetrahedra is changed to store the index of the fan tetrahedron again. Starting with all tetrahedra that are opposite to q of all fan tetrahedra, a final traversal over the mesh replaces all vertices p with vertex q where the traversal may not cross a fan tetrahedron.

5.3.2 Construction

A preprocessing step simplifies a tetrahedral mesh by a sequence of half-edge collapses $ecol(p, q, p)$ where a half edge $h = (p, q)$ is collapsed into the vertex p. Similar to predictive tetra meshes of § 5.2, a balanced hierarchy is constructed by collapsing independent edges only.

A binary hierarchy encodes dependencies between half-edge collapses and is used at run time to adapt the mesh to current viewing parameters. The hierarchy stores a node for each collapsed half edge. A node h_i with corresponding half-edge $h = (p_i, q_i)$ is parent of another node h_j with half-edge $h = (p_j, q_j)$, if $p_i = p_j$ and $j < i$, i.e. h_j has been collapsed before h_i. The nodes are enumerated in the order of their half-edge collapses such that half-edge collapses gets ascending indices.

In contrast to binary vertex hierarchies, a half-edge hierarchy contains half as many nodes and thus consumes fewer memory for representation as shown in figure 5.12.

The hierarchy consists of a forest of binary trees. Each node represents a half-edge which can be collapsed or split. In the following, the terms half-edge and node are used intertwined. If a half-edge is collapsed, its corresponding node is also said to be collapsed. Similar, if a half-edge is not collapsed, its node is also said to be not collapsed. Both, node and half-edge, are denoted with h.

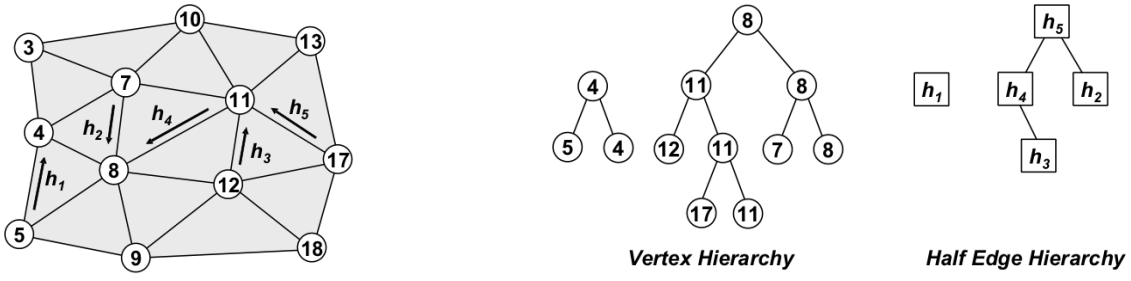


Figure 5.12: Instead of using a classic vertex hierarchy, we follow the FastMesh approach and create a half edge hierarchy for a sequence of edge collapses.

Each node of the hierarchy stores

- The half-edge h as index pair $h = (t, l)$ where t is the index of an tetrahedron incident to h while l selects the half edge within the tetrahedron t.

- The field error introduced by applying the half-edge collapse.
- The radius of a sphere that tightly bounds the modified region.
- The links to its parent and both children.

The index pair allows the half-edge h which corresponds to the node to be uniquely identified and to be collapsed or split. The field error and the bounding sphere are used to steer the adaptation of the mesh during rendering as described soon.

The index pair needs 5 bytes storage space, the field error needs 4 bytes, the radius needs 4 bytes, and the links need 8 bytes. So, the overall data structure needs 21 bytes per update.

5.3.3 View-Dependent Meshing

A valid mesh can be obtained from the base mesh by splitting half-edges and preserving all dependencies encoded in the hierarchy. An edge can be present in the mesh if and only if its parent edge is also present. Conversely, a parent edge can be collapsed if and only if its children are collapsed (not present in the mesh). In summary, a front of nodes through the half-edge hierarchy defines a valid mesh iff:

1. The front partitions the hierarchy horizontally such that the nodes above the front are currently not collapsed while the nodes below the front are currently collapsed. All half-edges of nodes exactly on the front can or cannot be collapsed in the current mesh.
2. Every possible path from the roots to the leaves contains just one node of the front.

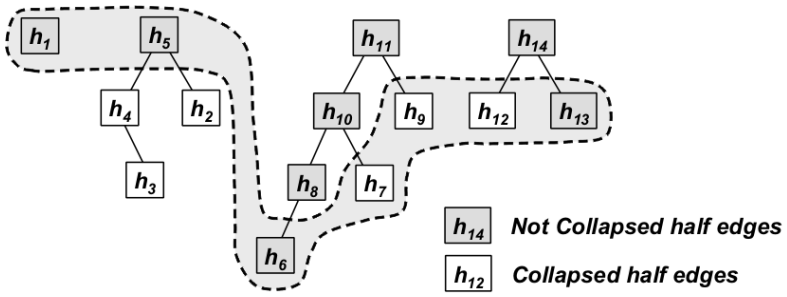


Figure 5.13: An edge front through this hierarchy defines a valid mesh if it partitions the nodes into nodes that represent collapsed edges and nodes that represent not collapsed edges. The front is implemented as a split and a collapse queue.

By traversing the node front, edges can be identified that need to be created or collapsed. Depending on the state of the edge corresponding to a node, the mesh can be refined or coarsened as follows:

- The edge is present (not collapsed).

  - Refine the mesh by splitting the children (which are currently collapsed according to the above definition), or
  - Coarsen the mesh by collapsing the edge.

- The edge is not present (collapsed)
  - Refine the mesh by splitting the edge.

After changing the state of a node, the edge front needs to be updated such that it fulfills criteria (1) and (2) again. The front is implemented as two queues: a refining queue and a coarsening queue. The refining queue contains all nodes that can (potentially) be split whereas the coarsening queue contains all node that can be collapsed.

The collapse and split of a half-edge h has been described above in § 5.3.1. Note that the information stored in a hierarchy node is sufficient to compute a collapse or split because one fan tetrahedron and its half-edge can be identified. The collapse and split can be computed starting at this tetrahedron.

One final problem needs to be solved. During a split, a fan tetrahedron can reference an adjacent tetrahedron that is not active, i.e. that has been deactivated by another half-edge collapse. If such a tetrahedron is not active, it must be made active by a forced vertex split. To ensure this, each tetrahedron stores the edge collapse that deactivated it. The forced vertex split may cause nodes to be split that have collapsed ancestors in the hierarchy. This would violate the conditions for a valid mesh. Thus, all ancestors must be split which can be accomplished efficiently by recursively splitting the parents.

The vertex split does not depend on a particular traversal of the tetrahedra that are incident to the split vertex nor needs a lexicographic sorting of the incident tetrahedra. It just uses the stored incidence information of the tetrahedra that are collapsed by the edge collapse and can traverse the incident tetrahedra in any order.

The validity of the vertex split operation is ensured by forcing the incident tetrahedra to exist. This is similar to Hoppe's condition of existing faces for triangle meshes [Hop97].

The validity of an edge collapse is checked explicitly at run time. Modified tetrahedra are checked for flipping and topological validity. This ensures that edges can be collapsed as long as the current mesh makes it possible. Note that due to the restriction to half-edge collapses, only those tetrahedra incident to q must be checked for a half-edge collapse $ecol(p, q, p)$.

Thus the mesh can be refined and coarsened very fast (for both vertex splits and edge collapses) and adaptively (for edge collapses). Remember that the simplification collapses independent sets of edges and thus reduces the average length of the forced splits.

We integrated the multi resolution model into a visualization framework that supports direct volume rendering with projected tetrahedra [ST90] and pre-integration [KQE04]. Initially, the model is loaded into memory with all nodes of the hierarchy collapsed. Thus the mesh is represented by its coarsest approximation. The indexed data structure contains all tetrahedra and vertices as they

have been left by the simplification process. The edge front is initialized to contain the root nodes only. All collapsed tetrahedra are marked as inactive.

In order to ensure interactive frame rates we want to find the mesh for each frame that best approximates the underlying classified scalar field up to a user-specified error tolerance and that is the nearly minimal mesh that has this error tolerance.

We use two priority queues that sort all valid edge collapses into one queue and all valid vertex splits into the other queue similar to Cignoni et al. [CDFM$^+$04]. All nodes of the edge front are stored in the queues. The priority pr of a node $h = (p, q)$ depends on a user-specified error threshold τ:

$$pr(h) = \alpha(h)e(h) - \tau$$

where $\alpha(h)$ is the largest alpha-value of all vertices in the link of the vertex p, including p, and $e(h)$ is the field error that is stored at h.

Given a node h from the split queue, a positive value of $pr(h)$ means that h should be splitted, a negative value means that h can be left unchanged. Positive values have higher priority. Given a node h from the collapse queue, a negative value means that h should be collapsed and a positive value means that h can be left unchanged. Negative values have higher priority.

Given furthermore a maximal number T of tetrahedra that an adapted mesh shall maximal contain, the mesh is adapted in a loop that works on the two queues in the following order.

1. If the current mesh contains more than T tetrahedra, the best node of the collapse queue is collapsed.

2. If the best node of the collapse queue has a negative value, the node is collapsed.

3. If the best node of the split queue has a positive value, the node is split if the mesh contains no more than T tetrahedra after the split. This is approximated by looking how many splits are needed to be forced and assuming that every split activates about 6 tetrahedra.

4. If the split would introduce to many tetrahedra, we collapse edges from the collapse queue as long as the mesh is coarsened enough to perform the split.

5. If the best node of each queue can be left unchanged (negative value for the split queue and positive value for the collapse queue), the adaptation stops.

The priorities are furthermore modified to support view-frustum culling by always setting $pr(h)$ to negative values if h belongs to the collapse queue and to positive values if h belongs to the split queue.

The screen projection error that is typical for multi resolution representations for triangle meshes can also be integrated by scaling the priority $pr(h)$ with the size of the projected bounding sphere of the node h.

Our dynamic meshing creates a conform mesh that is sent down a standard volume rendering pipeline. The renderer sorts the tetrahedra of the adapted mesh by depth in log–linear time and

Name	Bucky Ball	Blunt Fin	Sea	Fighter
# Vertices	32,767	40,960	102,165	256,614
# Tetras	176,856	187,395	524,640	1,403,504
MHD	13	15	20	17
# Tetra in base mesh	10K	10K	45K	49K
# Updates	30K	38K	250K	500K
Preproc. Time	50 sec	53 sec	154 sec	347 sec
Memory footprint [MB]	7.6	8.2	26.2	66.1
File Size [MB]	7.6	8.2	26.2	66.1
File Size Orig [MB]	3.2	3.5	12.1	25.9
Time Base→Orig	0.93 sec	1.02 sec	2.59 sec	7.1 sec

Table 5.2: The properties of the datasets. MHD is the maximal hierarchy depth. Time Base→Orig reports the time of extracting the original dataset from the base mesh.

sends the tetrahedra to a vertex shader program [WMF02] that projects the tetrahedra and renders the resulting triangles.

5.3.4 Results

The technique has been implemented and tested on a Pentium 4 with 2.8 GHz and 1 GB memory. The properties of the multi resolution models used for testing are shown in table 5.2.

At runtime an edge collapse takes about 0.02 ms such that we can remove up to 50K collapses per second. A vertex split takes about 0.015 ms due to the unnecessary check for validity. About 60K splits can be performed per second. Typically, algorithms that handle uniform volume data only [MDM04] can perform collapses and splits faster, but are restricted to uniform data. Our algorithm is aimed to handle unstructured data.

The pure renderer handles up to 450K tetrahedra per second (including sorting). Together with the dynamic meshing, we can render about $250\text{K}-400\text{K}$ tetrahedra per second depending on how many vertex splits and edge collapses are necessary to adapt the mesh.

The hierarchy needs 21 bytes per update. But the mesh itself needs to be stored in the main memory and needs 12 bytes per vertex and 32 bytes per tetrahedron. Finally, the index pair which is stored with each tetrahedron to find the edge that caused its collapse needs additional 5 bytes.

Figure 5.14 shows two scenes of an inspection of the Sea dataset together with a direct volume rendering. The observer sees all red-marked areas. The boundary triangle mesh helps to visualize the various resolutions of the mesh. Note the high resolution of the mesh currently visible to the observer in contrast to the coarse resolution of the mesh in all areas currently not visible to the observer.

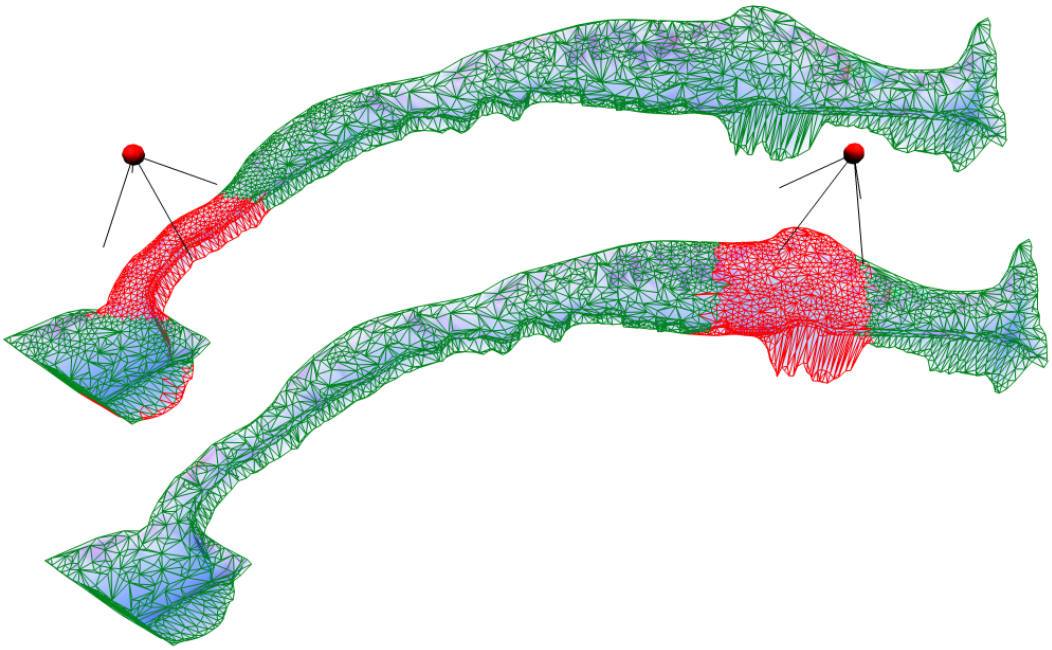


Figure 5.14: About 70K tetrahedra of the Sea dataset are inside the view frustum of the observer and need to be rendered instead of all 500K tetrahedra of the full resolution dataset.

5.4 Discussion

We have introduced two new multi resolution models for unstructured tetrahedral meshes. While Predictive Tetra Mesh allows for full edge collapses, FastTetraMesh extends the ideas of FastMesh from triangle meshes to tetrahedral meshes resulting in an extension of common data structures to multi resolution representations.

Memory Consumption

In terms of memory consumption, Predictive Tetra Mesh maintains the geometry and connectivity of the *adapted* mesh in main memory together with a binary hierarchy consisting of all of its nodes. In contrast, FastTetraMesh must maintain the geometry and connectivity of the *whole* mesh together with a binary hierarchy in main memory.

Nevertheless, the size of the binary hierarchy of FastTetraMesh is smaller because each node must store much less information than a hierarchy node of Predictive Tetra Mesh as shown in table 5.3. This is due to the fact that a hierarchy node of Predictive Tetra Mesh must store split information explicitly whereas a node of FastTetraMesh has to store no split information at all. FastTetraMesh is able to extract this split information from the mesh connectivity alone.

Because FastTetraMesh must store the whole mesh in main memory, its memory footprint is

	Predictive Tetra Mesh	FastTetraMesh	Cignoni et al.	Danovaro at al.
Mesh	**Adapted**	**Full**	**Adapted**	**Adapted**
Geometry	12	12	12	12
Connectivity	173	173	173	173
Total [bpv]	**185**	**185**	**185**	**185**
Hierarchy Inner Nodes				
Tree Structure	2×4	2×4	2×4	2×4
Field Error	1×4	1×4	1×4	1×4
Radius of Influence	1×4	1×4	1×4	1×4
Position Prediction	$2 \times 3 \times 4$	–	$2 \times 3 \times 4$	$2 \times 3 \times 4$
Attribute Prediction	2×4	–	2×4	1×4
Split Information	6	5	8	4
Total [bpv]	**48+6**	**21**	**48+8**	**40+4**
Hierarchy Leaves				
Active Vertex, Parent	2×4	–	2×4	2×4
Total [bpv]	**8**	–	**8**	**8**

Table 5.3: The memory consumption of Predictive Tetra Mesh and FastTetraMesh is reported together with the comparable previous approaches of Cignoni et al. [CMRS03] and Danovaro et al. [DDFM$^+$05]. Both floating point types and integer types are assumed to consist of 4 bytes. All values are given in bytes per vertex (bpv). Note that $32bpt \sim 173bpv$, see also chapter 2.

bigger compared to the footprint of Predictive Tetra Mesh, see also tables 5.1 and 5.2. Predictive Tetra Mesh stores a base mesh together with the hierarchy on disc and needs nearly as many disc space as a single mesh (an uncompressed mesh, of course) while FastTetraMesh needs to store the whole mesh together with its hierarchy. Hence, the disc footprint of FastTetraMesh is larger than that of Predictive Tetra Mesh.

Table 5.3 lists the memory consumption of two previous approaches which are comparable to ours. Cignoni et al. [CMRS03] implement a full-edge collapse and can thus be compared to Predictive Tetra Mesh while Danovaro at al. [DDFM$^+$05] implement a half-edge collapse and can be compared to FastTetraMesh, see also section 3.2.3. Note the improved encoding for split informations of Predictive Tetra Mesh ($48 + 6$) compared to Cignoni et al. ($48 + 8$). Cignoni et al. achieve this consumption only by restricting the valences of all vertices to at most 64 while Predictive Tetra Mesh can encode some datasets with only $48 + 5$ bits as described in section 5.2.

Run Time Behaviour

Both Predictive Tetra Mesh and FastTetraMesh use the operations vertex split and edge collapse. They differ in how the validity of those operations is ensured at run time.

For vertex splits, both FastTetraMesh and Predictive Tetra Mesh force additional vertex splits if the topological neighborhood of a split vertex does not equal the topological neighborhood of this

	Predictive Tetra Mesh	FastTetraMesh
Edge Collapse	0.015	0.02
Vertex Split	0.015	0.015

Table 5.4: The run time behaviour of edge collapses and vertex splits for Predictive Tetra Mesh and FastTetraMesh is compared. The timings are given in msec. for a single vertex split / edge collapse.

split vertex at preprocessing time. So, the run-time of vertex split operations of FastTetraMesh and Predictive Tetra Mesh should be equal to each other. This could be verified by our implementations and is reported by table 5.4.

For edge collapses, Predictive Tetra Mesh rejects this operation if the topological neighborhood does not equal the neighborhood at preprocessing time. Here, FastTetraMesh is more generous because the validity of an edge collapse is verified at run time by an inspection of the topological neighborhood. So, FastTetraMesh allows more edge collapses to take place but needs slightly more time to compute this validity. Table 5.4 confirms this behaviour and lists the average time that an edge collapse needs.

Chapter 6

Multi Resolution Models II

A set is a Many that allows itself to be thought of as a One. - Georg Cantor

With the ever growing size of meshes, two major challenges are to be attacked. First, the mesh might not fit into main memory and out-of-core techniques must be established. Second, references between mesh elements might be non-local leading to mesh accesses with a poor spatial coherence. Because memory hierarchies of modern computers (and especially the part of caching systems) are highly optimized for accesses with both high spatial and temporal coherence, algorithms relying on random accesses perform poorly for meshes with a bad reference locality. The processing unit cannot run at its full speed due to memory delays which has become a major bottleneck caused by the increasing gap between processing speed and memory speed.

Both challenges are more intertwined than one might think. The only way to solve the first problem is to store the mesh partially in main memory. But if references between mesh elements point into parts of the mesh currently not loaded, those parts must also be loaded which either increases consumed memory or forces other parts to be removed from main memory. While the latter often ends up with heavy swapping of mesh parts and a dramatically decreased performance, the first one may end up with a demand of memory that exceeds available in-core memory. To avoid such effects, all references between mesh elements should be as local as possible, i.e. the mesh elements should have a high spatial coherence. An efficient out-of-core technique requires high spatial coherence for all mesh elements.

Out-of-core techniques can be distinguished by the elements they are working on which range from the smallest possible elements, vertices and triangles (or tetrahedra), to larger elements like segments of the mesh. A mesh can be transformed into a representation telling an algorithm when to load and – very important – when to release single mesh elements like triangles or vertices allowing for a very fine control over the loaded mesh elements. Such a representation is called a streaming mesh [IL05] and ensures that a mesh element is not released until its last reference appeared in the representation. A representation allowing for a more coarse control over loaded parts of the mesh first breaks up the mesh into segments consisting of small parts of the mesh [SS06c]. As a necessary condition, a single segment must fit into core memory for processing.

Streaming meshes either exploit the spatial coherence present in the data themselves [ILSS06] or have to re-order all mesh elements such that their spatial coherence increases. In contrast, the spatial coherence automatically increases when a mesh is broken into segments because the vertices and triangles (or tetrahedra) are re-indexed when they are assigned to their segments.

This chapter covers two issues. First, a new algorithm is introduced together with a data structure allowing for a robust and fast partition of huge tetrahedral meshes into segments of nearly equal sizes. Additionally, the data structure is shown to enable an out-of-core processing of huge meshes by pushing the granularity of streaming from vertices and triangles to whole segments. As a main advantage, the automatic re-sorting of mesh elements in a segment-based approach is combined with the computational power and simplicity of streaming meshes.

Second, two multi-resolution models are introduced based on the segmented data structure. Fine segments can be replaced by coarse segments and vice versa enabling rapid changes of the multi-resolution mesh and exploiting modern memory hierarchies and graphic GPU architectures very well. Thereby, the correctness (consistency) of the multi-resolution mesh is always ensured.

As a side effect, we get rid of the restriction to a specific simplification operation. Vertex-based models need edge collapses as basic simplification operations while segment-based models can use any simplifier as long as the boundary of a simplified segment is left unchanged. A rapid point based simplifier like the one presented in § 4.2 is as well as suitable as any edge collapse based simplifier.

6.1 Segments

This section examines methods to partition huge tetrahedral mesh into segments enabling both an out-of-core processing and the construction of multi resolution representations. We assume that the meshes store scalar and/or vector values at their vertices.

Basically, the segmentation must consider the underlying memory hierarchy and needs to support memory allocation algorithms of the operating system. In order to avoid defraction of main memory, all segments should have a similar number of vertices and tetrahedra, that is, the segments consume a similar number of bytes. If segments furthermore fit into a single page of memory, the caching behaviour can further be optimized [YLPM05].

How can a tetrahedral mesh be split into segments?

Given a mesh, region growing techniques could try to *cluster areas of similar attribute values* into segments thereby separating areas of distinct attribute values. However, many meshes store more than one attribute value at their vertices, e.g. the Sea dataset has a total of seven attributes attached to each of its vertices. For such meshes, attribute clustering does not have a global meaning anymore. Additionally, clustering is not suitable if attributes evolve over several time steps.

Spatial partitions divide the mesh based on geometrical information only. Thereby, a mesh is typically partitioned by an octree or by a regular grid assigning first all vertices to their segments followed by all tetrahedra. Such partitions can be computed easily, rapidly and robustly. *Adaptive*

partitions further exploit the mesh topology and are often designed as region growing approaches. They are funded on both geometry and topology resulting in segments that reflect complex shapes well. Nevertheless, they require multiple runs over the initial mesh which slows down the creation process.

Caused by their simplicity and power, the presented out-of-core data structure uses a spatial partition. As a motivation, the drawbacks of spatial partitions based purely on octree subdivision or grids are discussed first.

The density of vertices of a tetrahedral mesh can vary significantly over the mesh in contrast to most triangular meshes as can be seen in figure 6.1a. There are areas with a high density (and rather small tetrahedra) and areas with a low density (and large tetrahedra). Often, those parts of the volume domain have a high density of points (and tetrahedra) where the solver of differential equations needs a high spatial resolution in order to be numerical stable and to capture all desired physical features, while other parts can be represented with a lower spatial resolution.

An octree partitions the vertices of a mesh quickly and robustly. Its leaves reflect the density of points in the mesh. Nevertheless, the number of vertices inside the leaves can differ significantly from leaf to leaf due to the regular subdivision when a leaf splits into its eight children. So, some leaves contain almost no vertices whereas other leaves contain many resulting in unbalanced segments, see figure 6.1b.

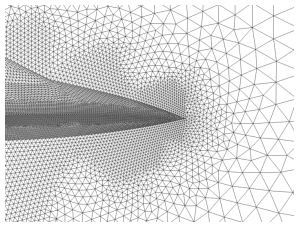
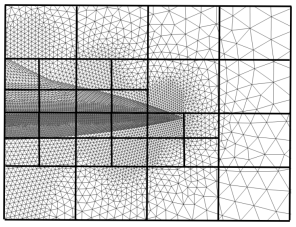
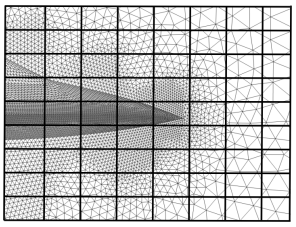
a)  b)  c) 

Figure 6.1: (a) The density of points in a tetrahedral mesh can vary significantly over the mesh domain as shown in the exemplary cutout of the Fighter model. (b) The leaves of an octree reflect the point density but contain an unbalanced number of vertices due to the regular subdivision. (c) A grid needs a resolution specified by the user and produces a highly unbalanced segmentation.

A regular grid has the disadvantage that a user must specify its resolution. Depending on this resolution, the variation of the number of vertices inside the grid cells differs highly. It is mandatory to merge cells into segments in order to obtain balanced segments. For tetrahedral meshes, the size of the grid cells needs to be very small in order to catch the fine-detailed areas of the mesh resulting in a vast number of grid cells that contain data. Nevertheless, a grid is well-suited as an out-of-core data structure because no grid cell needs to be subdivided (in contrast to octrees) and no vertex has to be re-accessed for subdivision purposes.

Additionally, when the cells of a grid are merged into segments, it is difficult to steer how the boundaries of the segments look like. If the user wants the boundaries to be horizontal, vertical or diagonal, the merging algorithm needs to consider this fact. Due to the fine-grained structure

of grid cells, the segments will most likely be well-balanced but the boundaries are not shaped in a way wanted by the user. Note that the shape of boundaries is a mandatory condition for some multi-resolution data structures as will be discussed in § 6.3. Furthermore, the merging algorithm scales linearily with the number of grid cells. Because many tetrahedral meshes need a very fine-grained grid, a merging algorithm does not only need much main memory but also slows down.

Name	Grid Resolution	# Segments	# Points / Segment		# Points / Segment			# Tetra / Segment		
			Minimal	Maximal	Minimal	Maximal	Average	Minimal	Maximal	Average
Sea	20x20x20	18	1	1,063	5,686	7,970	6,842	20,695	37,138	29,146
Fighter	20x20x20	47	1	15,515	1,648	17,424	8,418	6,510	89,171	40,100
Rbl	20x20x20	130	1	2,889	2,734	8,862	6,419	11,861	42,692	29,897
F16	20x20x20	212	1	700,456	1,122	700,456	4,090	4,138	3,935,947	22,867
F16	40x40x40	212	1	252,252	2,436	252,252	4,492	5,482	914,362	24,807
Earthquake	40x40x40	467	1	2,760	3,920	8,707	6,439	14,299	40,246	29,936

Table 6.1: Segmentation based on a grid with cell merging. The # Segments are given by the user. The first # Points / Segment gives the initial grid distribution while the second column gives the distribution after merging.

6.1.1 Out-of-Core Construction

The following approach combines the advantages of both octrees and grids. The produced segments have similar sizes in terms of their number of vertices and tetrahedra. The data structure can be computed very rapidly and the construction time depends linearily on the number of vertices and tetrahedra. The mesh is partitioned into segments by a first run over all vertices of the mesh followed by a second run over all tetrahedra. The first run creates an octree over all vertices. The leaves of the octree are merged into balanced segments which the tetrahedra are sorted into. As input parameters, the user specifies the maximal number K of vertices that are to be stored in each octree leaf and the final number of segments. (Alternatively, the user can specify the average number of tetrahedra per segment instead of the number of segments.)

The first run sorts all vertices of the mesh into an octree. If the number of vertices within an octree leaf exceeds the given number K, the leaf is subdivided at its midpoint into eight children. The vertices are stored in-core which is not critical because the number of vertices usually much smaller than the number of tetrahedra (in average by a factor of 6). After all vertices have been sorted into the octree, its leaves are merged to produce balanced segments. Therefore, an undirected graph is constructed whose nodes are the octree leaves and whose edges define which octree

Name	# Segments	# Points / Segment			# Tetra / Segment		
		Minimal	Maximal	Average	Minimal	Maximal	Average
Sea	80	52	5,446	1,640	60	24,179	6,558
Fighter	288	83	5,337	1,141	199	25,932	5,401
Rbl	687	2	5,518	1,889	1	26,518	8,431
F16	1128	1	5,312	1,761	20	23,412	9,591
Earthquake	2724	2	5,346	1,291	56	24,971	7,423

Table 6.2: Segmentation based on an octree without merging. The octree is steered to contain maximal 6,000 vertices in each of its leaves.

leaves are adjacent to other octree leaves. An edge exists between any two nodes if the boxes of the corresponding octree leaves touch each other at a box face. There exists no edge between nodes whose boxes touch only at a line or at a single vertex as shown in figure 6.2 which results in face-connected segments by a graph partitioning algorithm as described next.

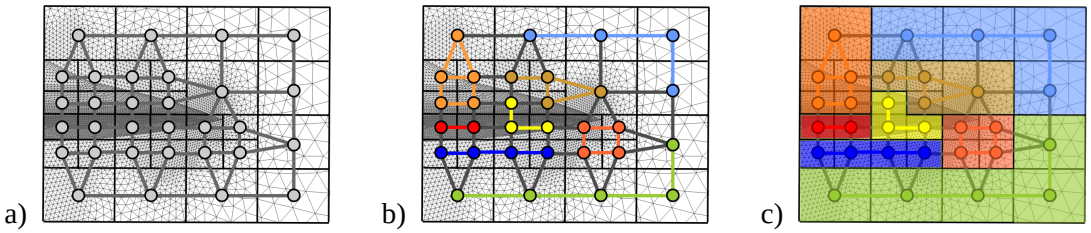


Figure 6.2: A graph connects the leaves (a) and is partitioned into balanced segments (b) which define the segments of the mesh.

A partition of a set S is a collection of subsets $S_i \subset S$ with $\bigcup_i S_i = S$ and $\bigcap_i S_i = \oslash$. The subsets S_i contain all elements of S and are disjoint, i.e. each element of S is contained in exactly one subset. In graph theory, a partition of a graph means a partition of its set of nodes. Often, the partition in question shall have a given number of subsets and fulfil some additional *constraints* as well as *objectives*.

A typical constraint requests the subsets to contain an equal number of nodes. If each node has an additional weight assigned to it, the constraint requests the subsets to contain nodes whose sum of weights are equal. The objective for such a constraint is to minimize the edge cut, that is, to minimize the number of edges straddling different partitions, or, if the edges have weights assigned to them, to minimize the sum of weights of straddling edges.

In order to find a partition of the constructed octree graph, the weight of each node is set to the number of vertices within the corresponding octree leaf while the weight of each edge is defined as the distance between the centers of the nodes connected by the edge. The partitioning routine computes a partition which balances the number of vertices while minimizing the edge cut. So, the objective favours compact and equal-sized subsets. Metis 4.0 [KK98, Kar98] implements very efficient partitioning routines for graphs and can be easily used by its programming interface.

The output of the partitioning routine is a collection of subsets each of which forms a segment and consists of a collection of leaves of the octree. Every leaf belongs to exactly one segment and thus every vertex of a leaf belongs to exactly one segment. An index is assigned to each segment. Because this index steers the assignment of tetrahedra as will be described soon, the index assignment needs to be done carefully. The segments are sorted ascending by the number of their vertices starting with the segment that has the fewest vertices. Each segment is assigned a number according to this ordering such that the first segment (with the fewest vertices) is assigned the lowest number.

Given the segments, a second run over the mesh assigns tetrahedra to their segments. Each of the four vertices of a tetrahedron belongs to exactly one segment. The tetrahedron is assigned to

the segment with the smallest index of all the four vertex segment indices. Note that segments with a low segment index get more tetrahedra assigned than segments with a high index as shown in figure 6.3 which improves the segment balance. The assignment works extremely fast because no computation needs to be done (neither computing additional points like centers of tetrahedra nor finding the segment that the additional points lie in, and no geometry information must be available at this point). Comparing to other techniques like [CMRS03] or [IG03] who assign a tetrahedron to the segment that its center lies in, no loss is perceived in the quality of how segments are shaped. Well-shaped borders between segments are created being fully sufficient for our purposes, see figure 6.3. For each segment, a file is created which the tetrahedra of the segment are written into. So, only the vertices are stored in-core.

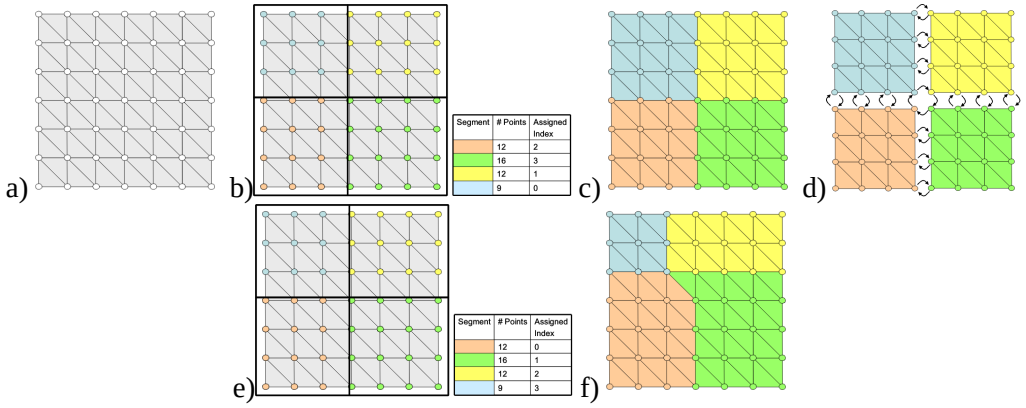


Figure 6.3: (a) An example triangular mesh. (b) The vertices are sorted into segments which are enumerated according to the number of vertices they contain. (c) The tetrahedra are assigned to segments based on the lowest segment index of their vertices. (d) Shared vertices have a copy in each segment they are referenced from. (e, f) A non-careful enumeration of segments results in un-balanced segments.

For simplicity of subsequent algorithms be it simplification or rendering, vertices that are shared between tetrahedra of different segments are copied for each segment. Such vertices are simply called *shared vertices* in the following text. As a tetrahedron is assigned to its segments and some of its four vertices belong to other segments than the tetrahedron's, each such vertex is copied into the tetrahedron's segment if the vertex has not been copied before. In order to keep track of copied vertices and to know if a vertex is shared between segments or not, the data structure described next stores an additional value for each vertex.

In order to establish terminology, all face triangles that a segment shares with another segments are called *segment boundary* between both segments, that is, the segment boundary is the cut between two segments. Note that the boundary of the tetrahedral mesh is not part of the segment boundaries.

A final step merges the tetrahedral files of all segments into a single file thereby deleting the temporary tetrahedral files. The out-of-core model is now ready for use.

Name	# Segments	# Points / Segment			# Tetra / Segment			Time	File Size
		Minimal	Maximal	Average	Minimal	Maximal	Average	[sec]	[MB]
Sea 1/30	18	5,344	8,976	7,075	19,308	38,020	29,146	4.40	21
Sea 2.5/30	18	4,121	11,641	6,901	17,232	45,382	29,146	4.38	21
Sea 5/30	18	4,310	12,602	6,653	18,046	60,335	29,146	4.08	21
Sea 0.5/15	35	2,835	6,115	3,888	10,074	23,352	14,989	4.9	22
Sea 0.2/3	175	460	2,146	906	513	7,064	2,997	6.6	23
Fighter 0.5/30	47	5,244	8,714	6,630	22,367	38,996	29,861	13.4	49
Fighter 1/30	47	4,919	8,158	6,536	21,083	40,290	29,861	14.2	49
Fighter 2.5/30	47	5,035	8,226	6,398	22,280	39,209	29,861	11.6	49
Fighter 5/30	47	3,639	10,123	6,330	16,838	49,762	29,861	11.39	49
Fighter 0.5/15	94	2,301	5,081	3,427	8,881	22,961	14,930	13.8	50
Fighter 0.2/7.5	188	1,348	2,843	1,859	3,824	11,097	7,465	16.7	50
Rbl 0.5/30	130	5,458	8,669	6,756	21,454	37,189	29,897	37.9	138
Rbl 1/30	130	4,860	8,913	6,455	21,284	38,088	29,898	38.9	137
Rbl 2/30	130	4,112	10,317	6,403	19,121	49,116	29,897	30.67	137
Rbl 5/30	130	2,347	10,212	6,342	10,748	49,591	29,897	31.05	137
Rbl 0.5/15	260	2,548	5,299	3,473	9,319	21,584	14,948	44.7	139
Rbl 1.5/15	260	1,448	5,185	3,366	6,121	22,485	14,949	33.5	139
Rbl 0.2/8	486	1,364	3,252	1,929	3,006	11,505	7997	64.1	140
Rbl 0.5/7.5	519	1,070	2,813	1,184	3,800	11,229	7,488	59.4	141
F16 1/30	212	4,618	9,477	6,554	19,060	43,930	29,932	70.0	246
F16 2/30	212	1,924	10,404	6,490	8,342	53,050	29,932	69.2	246
F16 5/30	212	1,094	11,169	6,522	4,428	56,953	30,362	69.4	246
F16 15/15	424	1,422	6,171	3,425	5,705	29,678	14,966	83.0	249
Earthquake 05/30	467	4,939	9,471	6,393	18,722	43,068	29,936	232	500
Earthquake 1/30	467	4,555	9,057	6,294	19,347	45,340	29,936	132	500
Earthquake 2/30	467	2,569	11,243	6,234	11,427	49,774	29,936	136	500
Earthquake 5/30	466	2,777	12,993	6,229	12,668	63,629	30,000	127	500
Earthquake 1/10	1399	2,777	12,993	6,229	12,668	63,629	30,000	189	524
Earthquake 05/7.5	1865	916	3,691	1,796	3,210	13,074	7,496	278	530

Table 6.3: Segmentation based on an octree with merging. The balance of the segments is achieved independent of the subdivision size. Sea 1/30 means that an octree leaf contains no more than 1K points and the leaves are merged to contain 30K tetrahedra in average.

6.1.2 Data Structure

Figure 6.5 shows the data structure that stores a single segment and contains an array of coordinates for vertices (`crds`) and an array of tetrahedra (`tetras`) together with their attributes (if any) and additional information useful for mesh traversal.

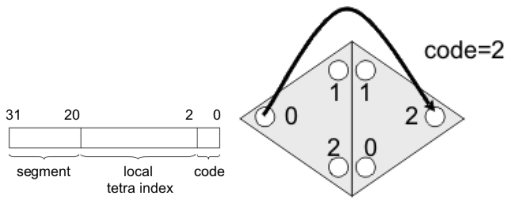


Figure 6.4: The opposite data structure can be implemented as a 32-bit field that stores a 12-bit segment index, a 18-bit local tetra index and a 2-bit code. The code specifies the location of the opposite point within the adjacent tetrahedron (shown as triangles for clearness).

For each tetrahedron four local vertex indices are stored comprising the `Tetra` structure. Because only local indices into the vertex list of the segment (`crds`) must be stored, a total of 16 bits is sufficient for each vertex index resulting in a total of 64 bits per tetrahedron (in contrast to

other representations that need twice as many space, namely $4 \times 32 = 128$ bits).

Each tetrahedron is face-adjacent to (at most) 4 other tetrahedra which is a necessary information for mesh traversals. The face-adjacencies are stored in the `opposites` array. For each face f of the four faces, `Opposite` stores the index of its adjacent tetrahedron and an additional code describing which of the four faces of the adjacent tetrahedron is adjacent to f. A 32-bit field is sufficient to hold all this information as shown in figure 6.4.

The data structure `VertexIndex` allows to identify vertices in a global unique way. It consists of an index-pair (s_i, l_i) with a segment index s_i and a segment-local index l_i which can be packed into a 32-bit field. Similar, a `TetraIndex` bit field allows for a global unique index for tetrahedra with l_i as the segment-local tetrahedron index.

Finally, the `shared` array keeps track of shared vertices as a single-linked list of `VertexIndex`es. For each vertex of the segment, the global `VertexIndex` of the next vertex that equals this vertex is stored as shown in figure 6.3. If the vertex is not shared, `VertexIndex` contains the index of the vertex itself. So, shared vertices can be identified easily by just comparing the stored `VertexIndex` with the current vertex' `VertexIndex` and all copies of a shared vertex can be iterated quickly.

```
Point * crds; // vertex coordinates
float * vtxattr; // vertex attributes
Tetra * tetras; // a Tetra stores 4 local vertex indices
float * tetattr; // tetrahedral attributes
Opposite * opposites; // face-adjacencies
VertexIndex * shared; // vertex index of the next equal vertex
```

Figure 6.5: The data structure for a single segment stores the coordinates and attributes for each point, the tetras and tetra attributes, the face-adjecencies between tetrahedra and an entry of the single linked list of shared vertices.

Additionally, each segment can store the indices of its *adjacent segments* which are important to be known because efficient streaming mesh processing can be accomplished by their knowledge as will be detailled next. A segment S_1 is *adjacent* to another segment S_2 if any mesh element of S_1 is adjacent to a mesh element of S_2. For instance, a vertex can be adjacent to a tetrahedron. Note that this relation is symmetric, i.e. if a segment S_1 is adjacent to another segment S_2, S_2 is also adjacent to S_1. A segment that is adjacent to another segment is also called a *neighbor* of this segment.

The segments are stored in a single file one after another in an order optimized for a streaming representation as described next.

6.1.3 Streaming segments

Basically, a *streaming data model* is a computational model which assumes that data elements arrive in a continuous sequential order. Algorithms have access to data elements that the stream has presented so far but do not have access to elements that will arrive in future. This does not mean that an algorithm does not know what happens in future but that all his knowledge must have its roots in data elements delivered so far.

Streaming data models distinguish offline and online models. An offline streaming model has control over the speed of the stream (arrival of data elements) stepping on at a pace that can be defined by the offline algorithm itself. An online streaming model must adapt to a speed of the stream defined by some external device like sensors or networks. Here, an algorithm must be able to consume all data elements at a given throughput.

The arriving elements are typically stored in a buffer enabling the algorithm to access them. A very important property of a streaming model is how the size of this buffer can be controlled. As data elements continue to arrive the buffer gets larger and larger growing to an infinite size if no data elements are deleted. So, the information about data elements that will *not* be accessed in future is essential for controlling the size of the buffer. This information can be either given implicitly by the application type or explicitly within the stream. For example, an application might filter the stream consuming a fixed number of input data elements which implicitly restricts the size of the buffer to at least this number of data elements. In contrast, meshing applications (like mesh smoothing or simplifying) need to store mesh elements in the buffer as long as they get accessed.

A data element is called an *active data element* if it is stored in the buffer. The number of all active data elements at a time t is called the *workload* at time t. The maximal workload of all times t is a critical value because it bounds the maximal number of memory that is used. A well-designed streaming model tries to minimize the workload.

As data elements are removed from the buffer, they may be discarded or written into an output stream (which can be simply a hard disc).

The workload is steered by two subjects. First, the order of data elements within a stream impacts the workload directly. Second, access patterns between data elements themselves determine which data elements need to remain active. To minimize the workload, the accesses between data elements should not spread over the whole stream but should be located close to each other. Otherwise, a data value has to remain active for a long time populating the buffer until it is be accessed for the last time and can be removed safely.

Streaming Model

As the basic idea of my streaming model, an algorithm processes a mesh segment-wise working on a single segment until it is fully processed and the algorithm proceeds with the subsequent segment. A streaming representation consists of a sequence of segments presented to the algorithm. Thereby the adjacency information between segments is exploited in order to minimize the

workload. Processing a segment may require access to other adjacent segments. For instance, a simplifier needs access to tetrahedra of another segment if an edge that is shared by two segments is collapsed.

If a segment is active, all its adjacent segments must also be active because their mesh elements may be referenced. So, segments that are adjacent to each other must be located close to each other within the stream. The stream consists of a sequence of tags which come at two types. A tag of the first type causes a segment to be loaded into the buffer while a tag of the second type specifies a segment to be removed from the buffer. The segments are requested in the order defined by the tags.

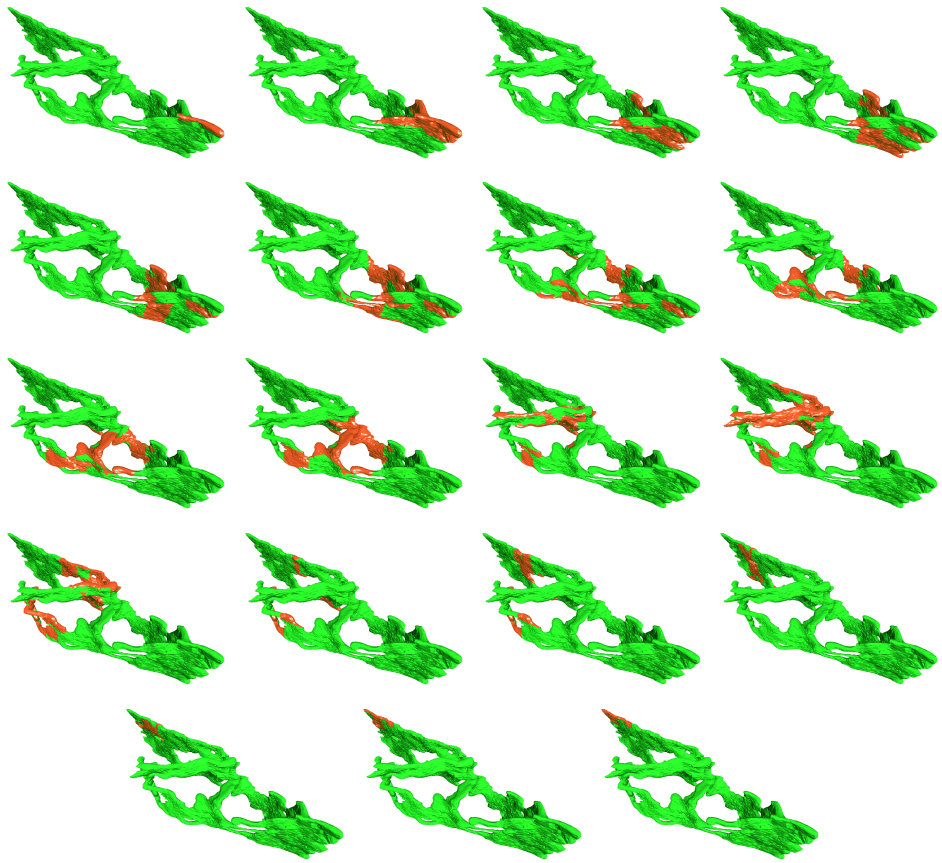


Figure 6.6: Streaming simplification of 3 million tetrahedra of the Rbl dataset. Red segments are stored in main memory, green segments are not.

A load tags is encoded as a positive number while a discard tag is encoded as a negative number. A typical stream looks like `1 4 5 7 -2 8 -6` where a positive number p means to load segment p while a negative number n means to release segment $-n-1$. If a released segment has been modified, it is written back to disc, and discarded otherwise. Given its index, a segment can be loaded from disc or network connections.

My streaming model maintains a list of active segments that are currently stored in main memory. An algorithm processes the segments in the order given by the load tags. In the example above, the stream steers the algorithm to operate on segments `1`, `4`, `5`, and `7`, release segment `1`, proceed with segment `8` and release segment `5`.

In order to create the stream, only the adjacency information between segments needs to be known and can easily be computed at construction time of the segmented mesh. So, the stream can be created very quickly even for huge meshes and all mesh processing algorithms like simplifiers or compressors can be easily modified to work on such a stream of segments.

Creation

In order to create the stream, the segments must be ordered such that the workload is minimized. Basically, the order corresponds to a walk over the adjacency graph of the segments. Therefore, we count the number of neighbors for each segment and store the counts in a simple array. The count holds information about how often a segment can be referenced by other segments and enables us later to decide if a segment can be released.

We maintain the stream itself and a candidate list which contains all segments that are candidates for insertion into the stream. As long as the candidate list contains segments, we select an optimal segment from the list, add its load tag to the stream and remove it from the candidate list. If a load tag of a segment is added to the stream, two things happen. First, for each of the segments neighbors, the stored count is decreased by 1. Second, all neighbors are added to the candidate list if they are not contained in either the stream or the candidate list. If the count of a segment reaches 0 and its load tag has been added to the stream before, a release tag is added to the stream for this segment.

The optimal segment is chosen to keep the workload small. We select the segment which needs to add the fewest number of segments to the candidate list. So, the size of the candidate list is kept small exploiting the adjacency coherence. A small size of the candidate list increases the likelihood that a segment can be released. A segment is released if its count reaches zero. Only other segments can reduce the count to zero. Because the adjacency relation is symmetric, a segment can be released only if all its neighbors and the segment itself are processed which is forced by a small candidate list.

The stream is constructed by starting with the segment whose bounding box is closest to the minimal point of the whole dataset's bounding box. (We could also start at some other extremal point of this bounding box.) This segment is added to the list and the selection process starts as described above.

Results

Given the segmented models of § 6.1, the meshes can be processed with a maximal workload as reported by table 6.4. As applications, the adjacency information of the meshes are calculated and

the meshes are simplified using quadric error metrics. The timings are reported in table 6.5. They were measured on a Pentium 4, 2.8 GHz with 1 GB main memory.

Name	# Segments	max segs [%]	max mem
Sea 1/30	18	66.7	20
Sea 2.5/30	18	55.5	16
Sea 5/30	18	61.0	17
Sea 0.5/15	35	47.2	13
Sea 0.2/3	175	25.5	7
Fighter 0.5/30	47	57	35
Fighter 1/30	47	51	30
Fighter 2/30	47	53	32
Fighter 5/30	47	51	31
Fighter 0.5/15	94	44	27
Fighter 1.5/15	94		
Fighter 0.2/7.5	188	44	27
Rbl 0.5/30	130	24.2	38
Rbl 1/30	130	22.8	36
Rbl 2/30	130	23.1	36
Rbl 5/30	130	23.0	36
Rbl 5/15	260	20.0	33
Rbl 1.5/15	260	19.2	31
Rbl 0.2/8	486	19.8	57
Rbl 0.7/7.5	519	20.6	33
F16 1/30	212	37	98
F16 2/30	212	39	103
F16 5/30	212	35	92
F16 1.5/15	424	31	87
F16 1.5/7.5	845	26	79
Earthquake 05/30	467	24	133
Earthquake 1/30	467	24	133
Earthquake 2/30	467	24	133
Earthquake 5/30	467	22	123
Earthquake 1/10	1399	17	98
Earthquake 05/7.5	1865	13	82

Table 6.4: Streaming properties of the segmentations of table 6.3.

Name	Time Adjacency Calculation [sec]	Time Simplification to 10% [sec]
Sea 5/30	2.0	56
Fighter 5/30	4.7	187
Rbl 5/30	13.0	397
F16 5/30	28.7	609
Earthquake 5/30	56.9	914

Table 6.5: Timings for constructing the adjacency information and edge-based simplification.

6.2 Binary Tetrahedral Segment Hierarchies

This section describes a multi-resolution model for large tetrahedral meshes which is based on the segmented representation discussed in the previous section. A binary *hierarchy of segments* is constructed where each node represents a single segment. The leaves represent the segments of the original mesh while inbetween nodes represent coarser and coarser simplified segments. The section starts with a description of the construction of the multi-resolution model, followed by a discussion of its usage and integration into a rendering system.

6.2.1 Out-of-Core Construction

The hierarchy of segments is constructed bottom-up level by level. The lowest level is created by duplicating all segments of the original mesh. Each duplicate is simplified leaving its segment boundaries unchanged, i.e. all shared vertices between segments are left untouched. We end up with a coarse approximation of each segment where different segments are still connected at the original resolution level along their segment boundaries. The lowest level of the binary multi-resolution hierarchy is constructed by inserting the simplified copies as parents of their original segments as visualized figure 6.8.

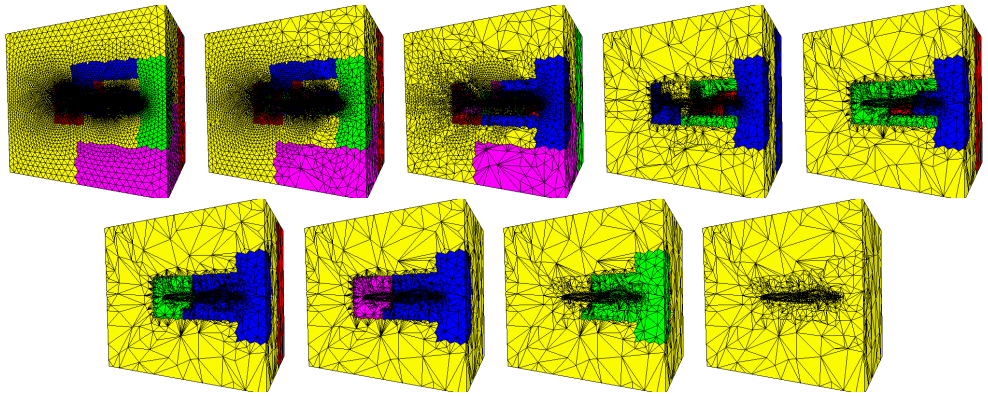


Figure 6.7: The construction process for the NASA fighter dataset merges two segments into a new segment which is simplified. The colors of the segments can change from picture to picture.

Next, the binary hierarchy is grown upwards level by level. For each level, an undirected graph is constructed whose nodes are the simplified segments of its next lower level. An edge exists between any pair of segments that share at least one tetrahedral face. The nodes are weighted by the number of vertices in their segments. A matching of the graph nodes is computed and ends up with pairs of nodes (segments).

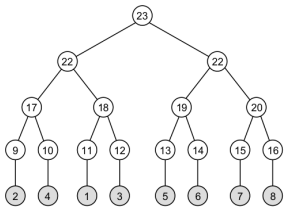


Figure 6.8: The binary segment hierarchy stores how the segments are merged into their parent segments. The segments of the original mesh are gray.

Each segment pair is merged into a new segment and a new level is added to the binary multi-resolution hierarchy by inserting the new segment as parent of both its children as shown in figure 6.8. The new segment is simplified. Note that all vertices that have been shared between

both segments are now fully contained in the new segment. They are not shared vertices anymore which enables them to be simplified. In contrast, shared vertices on the segment boundaries remain shared vertices.

Figure 6.9 depicts the overall algorithm. Every iteration constructs a neighboring graph and computes a graph matching which results in pairs of segments. Both segments of a pair are merged into a new segment which is simplified. The hierarchy grows by building a (at most levels binary) tree where the new segment is the parent of both merged segments. Figure 6.7 shows the construction process for the NASA fighter dataset.

```
For every segment s
    s_new = copy segment s;
    add s_new as parent of s;
    simplify s_new;
While the number of roots > 1
    construct the node graph G_H;
    find pairwise matching M_H of G_H;
    For every pair (s_1, s_2) in M_H
        s_new = merge segments s_1 and s_2;
    construct the dependency graph G_D with edge weights R_{s_new1, S_new2};
    find pairwise matching M_D of G_H;
    For every new segment s_new
        add s_new as parent of s_1 and s_2;
        simplify s_new and all boundaries to any segment s_new2 with (s_new, s_new2) ∈ M_D;
```

Figure 6.9: Every iteration finds pairs of segments that are merged into a new segment and simplified.

The final mesh segments are written in the streaming order defined in section 6.1.3 to disc together with the binary hierarchy. Using a streaming order improves the timings of disc accesses because adjacent segments are likely to be stored near to each other.

The nodes near the roots in the hierarchy contain very coarse approximations of the mesh while the boundaries between segments are still at the original resolution resulting in a highly unbalanced point distribution for ever bigger meshes. Such an unbalanced point distribution leads to poor simplification results. Often, badly shaped mesh elements are created or the mesh can't be simplified further due to topological and geometrical constraints. So, the multi-resolution data structure allows the boundaries between segments to be simplified, too. The simplification of a boundary between two segments must be reflected in the binary multi-resolution hierarchy by adding further dependencies between nodes.

If a boundary between two segments is simplified, a *dependency* is added to the hierarchy. Dependencies are a symmetric relation, that is, if segment S_1 depends on segment S_2, segment S_2 also depends on segment S_1. As will be explained in the next section, dependencies must be

added carefully because they directly influence the number of segments (and thus tetrahedra) that need to be active in a view-dependent mesh.

The best strategy that the author could find for placing dependencies *without any threshold* provided by the user, is based on the ratio $R_{S_1,S_2} = \frac{|B|}{min(|S_1|,|S_2|)}$ between the number of faces $|B|$ on the boundary between two segments S_1 and S_2 and the number of tetrahedra $|S_i|$ within S_i, $i = 1, 2$. The ratio has a high value if the boundary is highly oversampled compared to the interiour of the segments while the ratio is low for a well-sampled boundary. A graph is constructed (again) with all segments as nodes and with edges between any face-connected segments. The edge between two segments S_1 and S_2 is weighted by the ratio R_{S_1,S_2} and a pair-wise graph matching is computed which favours edges with a high ratio. The boundary between segments of the pair-wise matching are allowed to be simplified and a dependency between both segments is added to the hierarchy. The pair-wise matching decreases the number of split-chains which would ruin the efficiency of the multi-resolution model.

6.2.2 Dynamic Meshing

Each node of the binary hierarchy corresponds to a small tetrahedral mesh. A collection of nodes is called a *segment front* and corresponds to a conforming tetrahedral mesh iff

1. From every path along nodes from the root to the leaves, there is exactly one node on the segment front.

2. For each node on the segment front, all its dependent segments are also on the segment front.

Item 1 ensures that the mesh which is formed by all segments of the segment front has no holes and covers the complete domain of the original mesh. Item 2 ensures that no two segments overlap and that the segments match seamless at their segment-boundaries. Figure 6.10a visualizes a valid segment front. A segment on the front is called an *active segment*. A segment front corresponds to the well-known vertex front in vertex-based multi-resolution models like [DDFM+05] or [SS05b], see § 5.2.

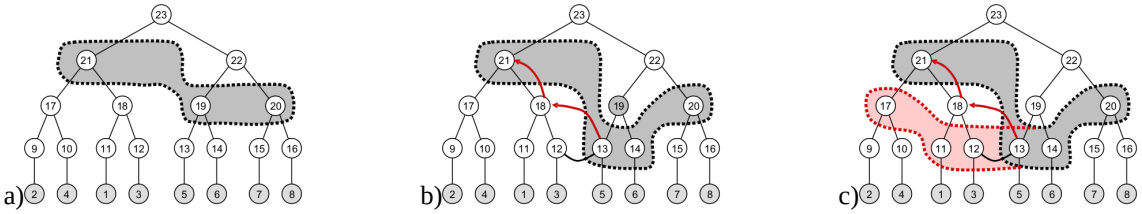


Figure 6.10: (a) A segment front through the binary hierarchy corresponds to a valid mesh. (b) If segment 19 is refined, the segment front must be made valid by additional refinements of dependent segments (thick black line between segments 12 and 13). (c) After forced refinements, the segment front is valid. The red area shows the segments that are forced to be active.

The segments on the segment front are candidates for refinement and coarsening, respectively. The refinement of a segment S replaces the segment S by its children S_1 and S_2 and ensures all dependencies of S_1 and S_2. If a child segment depends on another segment S_d, this segment S_d must forced to be active by refining its parent segment as shown in figure 6.10. By recursively refining parent segments, all dependencies can be ensured.

A segment can be coarsened if its sibbling as well as all its dependent segments are on the segment front. Then, both sibblings can be replaced by their parent segments and all sibblings among dependent segments can be replaced by their parent segments.

6.2.3 The 0-segment

The meshes of all segments on the segment front form a valid tetrahedral mesh. During rendering, the adjacencies between tetrahedra are often needed (for instance for MPVO-based sorting [Wil92]) and especially between tetrahedra of different segments. A fast method to adapt the adjacencies between segments is needed whenever a segment is replaced by another segment. We introduce a special segment that handles the adjacency between any two neighboring segments. Because this special segment has the unique segment index 0, we call it 0-segment.

Theoretically, the 0-segment contains all triangles that form boundaries between segments, that is, all triangles that result from cuts between all adjacent segments. Those triangles are used as a buffer (or a docking station) between two adjacent segments. Instead of storing the index of an adjacent tetrahedron, the index of the shared triangle in the 0-segment is stored in the face-adjacency information, see data member `opposites` in figure 6.5. Because the triangles never change and have always the same index, the adjacency to this triangle can be stored in all faces of tetrahedra that are boundary faces. Note that these indices need never to be changed. The 0-segment forms a special data structure and is the only data structure that must be changed as described next.

Each triangle stores two adjacency index-triples of type `Opposite` that point to tetrahedra in the adjacent segments, see also figure 6.11b. All tetrahedra of a segment having a segment-boundary face store an index-triple (s_i, t_i, c_i) that points into the 0-segment with $s_0 = 0$ and t_i as the triangle index. c_i can be 0 or 1 and points to one of the two adjacency index-triples of the triangle.

When a segment s replaces another segment r, it must update the 0-segment as follows. All tetrahedra of s having segment-boundary faces are traversed. At least one of the four adjacency index-triples of these tetrahedra point to a triangle t in the 0-segment (the other index-triples point to tetrahedra inside s itself). The index-triple of t points to the segment r and must be replaced by the index-triple of s, i.e. (s_i, t_i, c_i) where $s_i = s$, t_i is the index of the border tetrahedron and c_i is the local index of the opposite vertex within t_i, see also figure 6.11.

In order to find all border tetrahedra fast, they are stored before all other (inner) tetrahedra in the file. So iterating all boundary tetrahedra can be accomplished by iterating over all tetrahedra until a non-boundary tetrahedron is found.

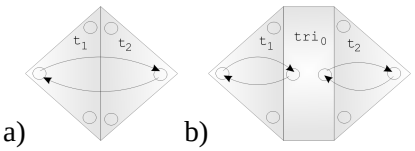


Figure 6.11: Instead of storing the index-triple of the neighboring tetrahedron at the border of two segments (a), the index of the triangle in the 0*-segment is stored (b).*

The 0-segment does not need to exist at the construction phase. It must exist only at run-time when segments are to be exchanged very fast and can be computed once after (or before) the construction phase.

6.2.4 View-Dependent Meshing

In order to adapt the mesh to current viewing and classification parameters, we need to decide which segments of the segment front must be coarsened (replaced by the parent) or refined (replaced by the children). Therefore, the following values are stored with each segment

- The histogram H of the attribute values (which is a lookup-table of resolution N with normalized entries $H_i \in [0, 1]$), and
- A look-up table E of the same resolution N which specifies the maximal error that the segment contains for the according attribute value, and
- The (axis aligned) bounding box.

The user can specify a maximal field error ϵ_{max}.

For each frame, the segment front is traversed. Every segment of the front is marked by the tags COARSEN, REFINE, and NOTHING that help later to adapt the mesh:

1. Compute the average sum $S = \sum_i^N H_i E_i \alpha_i$ where the sum runs over all histogram values, H_i is the (normalized) i-th histogram value, E_i is the according error and α_i is the according classified opacity.

   If $S > \epsilon_{max}$, mark as REFINE, else

2. If the bounding box is outside the view-frustum and if the sibbling is on the front as well as all dependent segments (of both the segment itself and its sibbling), mark as COARSEN, else

3. Mark as NOTHING.

After all segments of the front are marked, the front is traversed again and every segment is adapted:

1. If a segment is marked as REFINE, it is replaced by its children and all dependent segments are forced to be active as described above.

2. If a segment and its sibbling as well as all dependent segments are marked as COARSEN, they are replaced by their parents.

The histogram H as well as the look-up table E are computed during the construction of the multi-resolution model. Every edge collapse introduces a particular error for a scalar field attribute which is stored in the look-up table E if it is greater than the already stored error.

A simple LRU queue keeps track of all loaded segments. The queue contains at least all active segments but may also contain released segments.

6.2.5 Results

The segments are swapped from disc via memory mapped files which are an opportunity of modern operating systems for memory allocation. Parts of a file can be mapped directly into main memory. If a segment is requested, the part of the file which corresponds to the segment is mapped into main memory and can be interpreted immediately as the `points`, `tetras`, or similar arrays of § 6.1.2. Because the operating system exploits disc caches in a highly sophisticated way, memory mapped files map parts of a file in nearly constant time if the part has been accessed some time before. However, sometimes cache misses happen and small delays occur due to the time of positioning the disc read head. The memory of a mapped file can be released in a similar way to other memory allocation techniques. This way, the system can change up to 500,000 tetrahedra per second. Note that this value is highly dependend on the status of the caches and was measured as peak performance. In average, if segments need to be loaded for the first time, the performance decreases .

The frame rates depend mainly on the power of the visualization system. The volume renderer uses preintegrated projected tetrahedra. Because the size of the segment front is small, the costs for checking if a segment can be refined or coarsened, are neglectable. So the checking methods may be more expensive in calculation than those of per–vertex based multi resolution models. We experienced frame rates of about 3–4 frames per second with a workload of about 250,000 tetrahedra.

In average, 80% of the time of a frame is used by the volume renderer whereas 16% are used for reloading segments (file IO) and 4% are used for segment adjacency adaption (measured average values for the NASA Fighter dataset, the other datasets perform similiar).

The construction timings of the multi-resolution meshes are reported in table 6.6. The main time was spent to simplify the meshes and to construct the hierarchy. Although the timings for simplification do not compare to the (much faster) timings of Lindstrom [VCL+05], we differ from Lindstrom because we need to store all segments and do not use the randomized edge collapses.

Furthermore, the file sizes are huge because we store each segment such that it is ready to be mapped to memory.

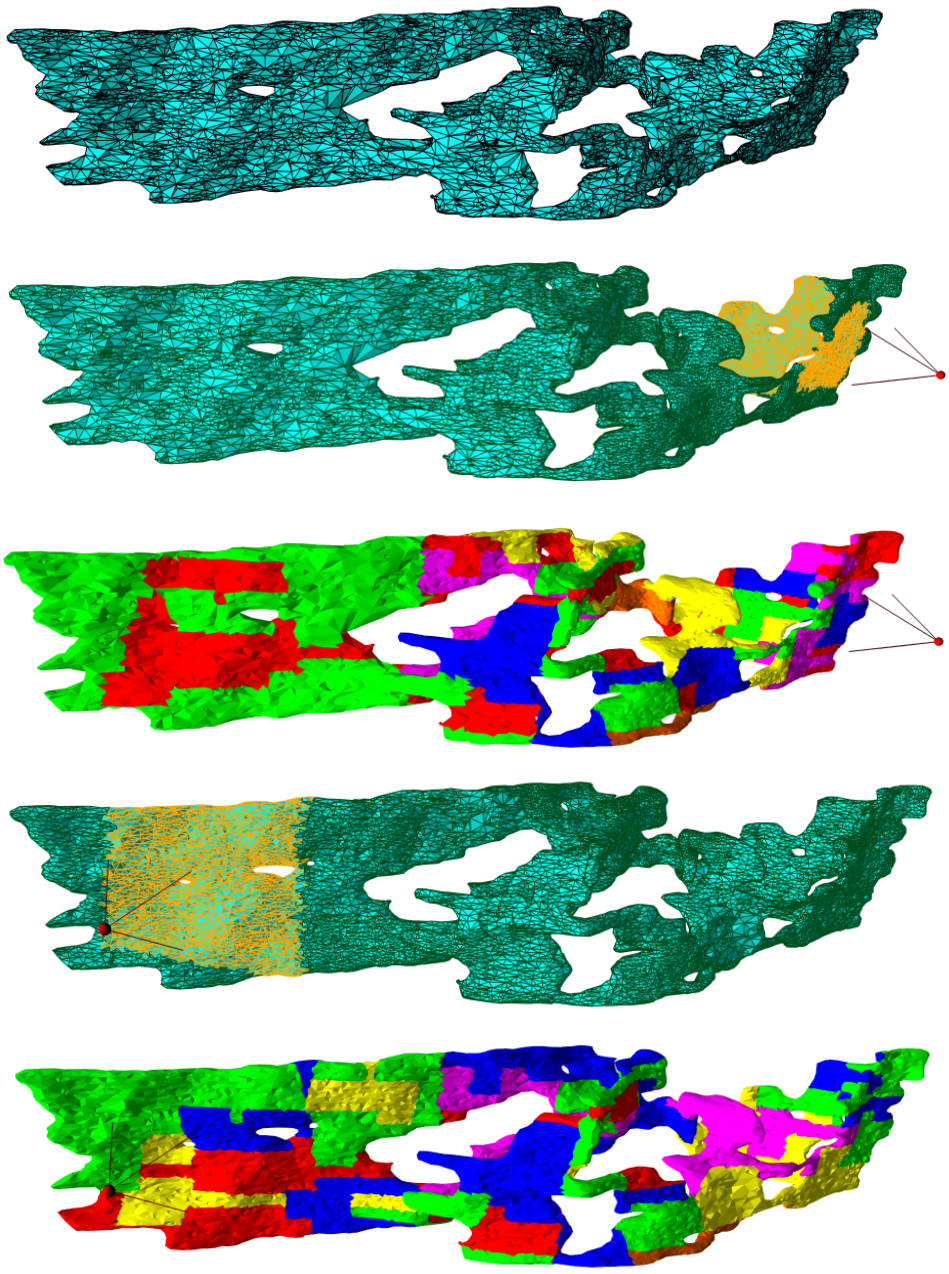


Figure 6.12: The base mesh of the Rbl dataset (top) contains about 80,000 tetrahedra and is adapted to an observers position (red sphere). The mesh consists of segments that perfectly match to each other along their boundaries.

Name	# Tetra in base mesh	Time hh:mm:ss	# Segments in hierarchy	Maximal Memory Usage [MB]	file size [MB]
Sea	15K	0:08:47	38	34	67
Fighter	30K	0:14:23	95	43	101
Rbl	80K	0:25:05	420	101	284
F16	80K	0:46:13	473	187	513
Earthquake	120K	1:23:21	967	300	1,100

Table 6.6: The construction properties including timings and the sizes of the stored multi resolution files. The timings include file I/O and simplification.

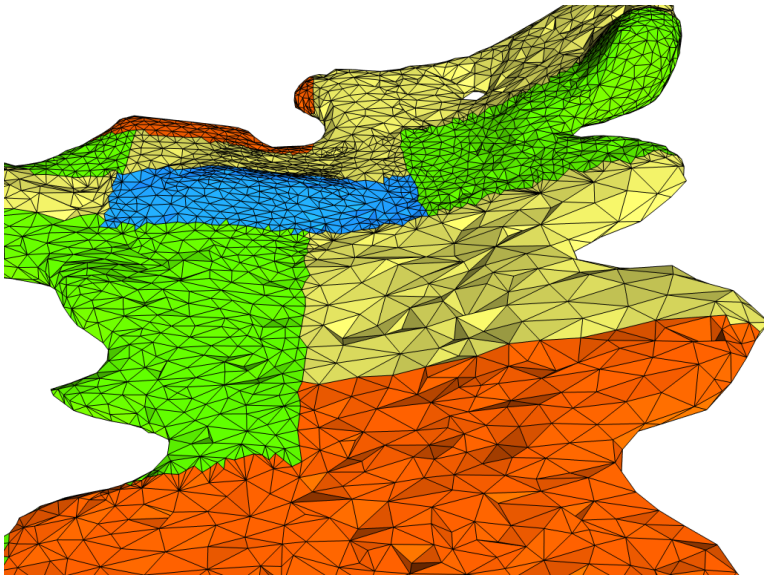


Figure 6.13: The meshing quality in a zoom into the Rbl dataset.

6.3 Tetrahedral Multi Triangulations

A very flexible n-ary hierarchy applies the concepts of multi-triangulations to tetrahedral meshes. A multi resolution model is constructed which automatically builds high-quality meshes in terms of both approximation quality and tetrahedral shape quality. All segments are balanced as much as possible, that is, they contain a similar number of tetrahedra and vertices which supports the memory management of the underlying operating system.

The first section overviews the concepts of multi-triangulations while the second section shows how the segmentation can be used to construct a multi-triangulation for huge tetrahedral meshes.

6.3.1 Basic Multi Triangulations

Multi Triangulations are a powerful tool to build multi resolution representations of simplicial complexes [KG98, Pup96]. [CGG+05] extended them to not only support simplicial complexes but also parametric surfaces or point clouds, and applied them to render huge polygonal models interactively. This section gives a brief overview of how multi triangulations for simplicial complexes work.

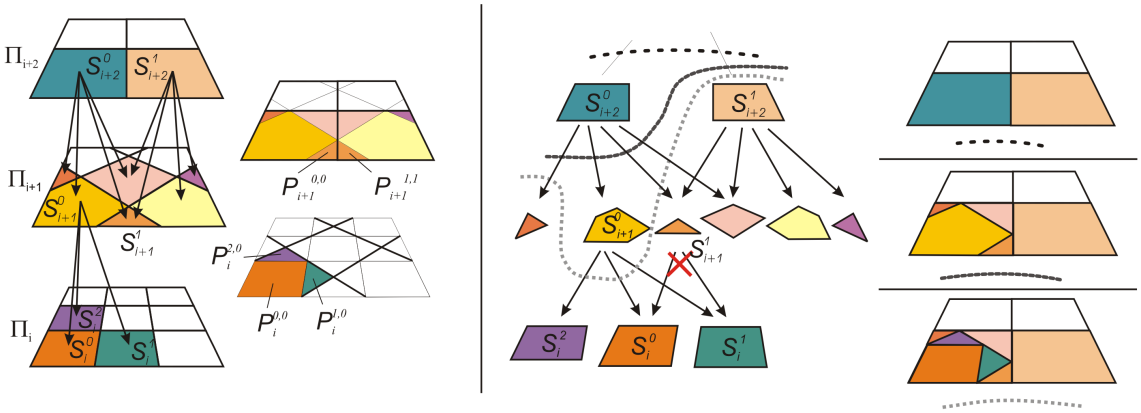


Figure 6.14: A 2D example. (Left) The multi triangulation is constructed by a sequence of partitions Π_i. Each partition consists of segments S_i^j that cover the whole mesh. The segments between two consecutive partitions are cut to form the patches $P_i^{j,k}$. (Right) A directed acyclic graph (DAG) describes the dependencies between the segments. The resulting segmentations of three different cuts through the DAG are shown. Note that the segment S^1_{i+1} may not be subdivided anymore in the third cut without additional subdivisions of its parent S^1_{i+2}. Otherwise, the patch $P^{1,1}_{i+1}$ would overlap the segment S^1_{i+2}.

Given an input mesh (triangular or tetrahedral), a multi triangulation is built by a sequence of partitions $\Pi_1, \Pi_2, ..., \Pi_n$ of the mesh. Each partition Π_i divides the mesh into a number of segments S_i^k, $k = 1, ..., |\Pi_i|$ that may only overlap at their boundaries as shown in figure 6.14 (left).

A multi triangulation consists now of *patches* $P_i^{j,k}$ which are defined as cuts between the segments of two consecutive partitions Π_i and Π_{i+1}:

$$\begin{aligned} \Pi_i \bigcap \Pi_{i+1} &= \bigcup_{j \in 1..|\Pi_i|, k \in 1..|\Pi_{i+1}|} (S_i^j \cap S_{i+1}^k) \\ &= \bigcup_{j \in 1..|\Pi_i|, k \in 1..|\Pi_{i+1}|} P_i^{j,k}. \end{aligned}$$

A *directed acyclic graph* (DAG) describes which segments cut each other. The nodes of the DAG are the segments S_i^j while all patches $P_i^{j,k}$ define the directed edges. Thereby, an edge exists between a so-called ceiling segment S_{i+1}^j and a floor segment S_i^k if both segments overlap.

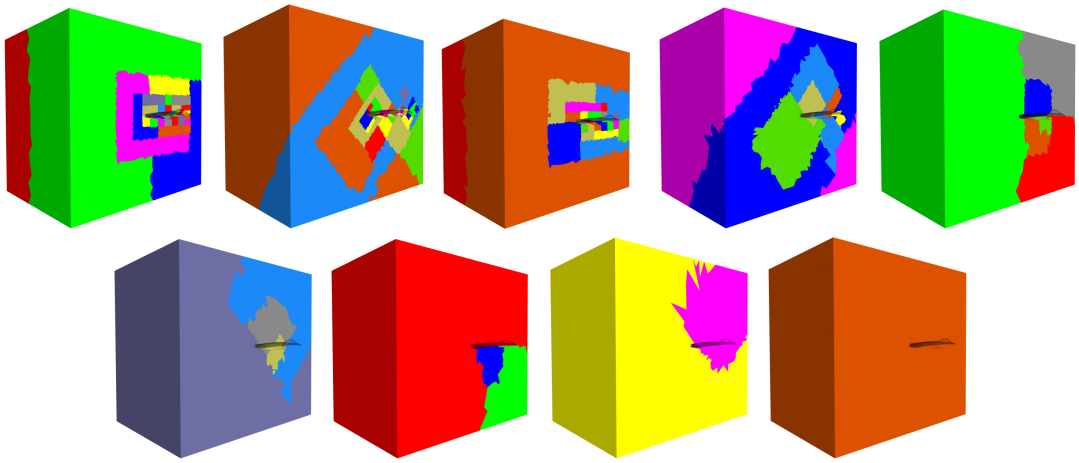


Figure 6.15: The constructed sequence of partitions for the Fighter dataset starts with an initial partition of 48 segments. The final partition consists of just one segment.

A *consistent mesh* is assembled by a list of patches that do not overlap (except on their boundaries) and cover the whole mesh. It can be expressed as a list of edges in the DAG, i.e. by a *graph cut*. Because the patches may not overlap, every path from the roots to the leaves must contain at most one cut edge. In order for the patches to cover the whole mesh, every path from the roots to the leaves must contain at least one cut edge. Combining both, every path must contain exactly one cut edge.

A segment S_{i+1}^k can be replaced by all patches $P_i^{j,k}$, $j = 1, ..., |\Pi_i|$, that have this segment as ceiling segment:

$$S_{i+1}^k \leftarrow \bigcup_{j \in 1..|\Pi_i|} (S_i^j \cap S_{i+1}^k) = \bigcup_{j \in 1..|\Pi_i|} P_i^{j,k} \tag{6.1}$$

On the other hand, if all patches $P_i^{j,k}$, $j = 1, ..., |\Pi_i|$, are on the graph cut, they can be replaced by their ceiling segment S_{i+1}^k as shown in figure 6.14 (right).

This concept works for a sequence of partitions that was constructed for just one mesh as well as for a sequence of partitions that is constructed by consecutively simplified versions of the mesh as described next.

The partitions are constructed iteratively starting at a partition Π_0 of the full-resolution mesh. Given a partition Π_i, we construct its consecutive partition Π_{i+1} as follows. The vertices and tetrahedra of the segments S_i^j in Π_i are distributed to the consecutive partition Π_{i+1} resulting in segments S_{i+1}^k. These segments are simplified whereby their borders are left unchanged. Afterwards, the iteration continues and the next partition Π_{i+2} is constructed from Π_{i+1}.

Because the borders of the simplified segments of one partition are left unchanged, the vertices and tetrahedral faces (triangles) on these cross-segment borders are contained in both the partition Π_i and its consecutive (simplified) partition Π_{i+1}. So, the refinement and coarsening operations of (6.1) can be performed and the constructed multi-triangulation forms a valid multi-resolution

model.

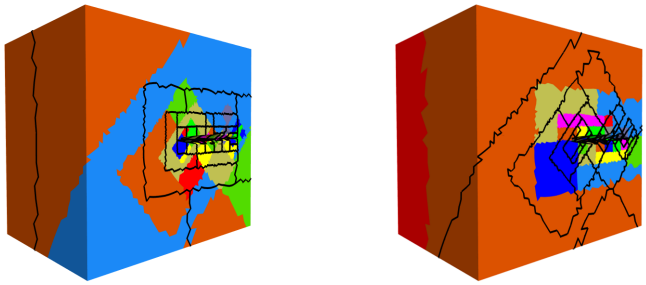


Figure 6.16: Two consecutive partitions of the Fighter model. The borders of the next partition are rendered on top of the segmentation of the current partition.

6.3.2 From the Basics to Tetrahedral Meshes

We need to design a sequence of partitions that creates balanced and well-shaped patches. Additionally, the border of a segment of one partition must be mostly contained in the segments of its consecutive partition because the simplification leaves the borders of segments unchanged. So, border vertices that have been left unchanged in one partition will be simplified in the consecutive partition.

The upcoming questions are: How shall the segmentation be chosen in order to create high-quality approximations and to enable an well–formed DAG, i.e. the dependency graph between segments must be balanced and do not contain long chains of dependencies. Such chains force segments to become active because of a dependency relationship and not because of a requirement of the visualization system.

Inspired by the bintree hierarchy [DWS$^+$97] and $\sqrt{2}$-subdivisions, we construct the sequence of partitions iteratively by rotating the segmenting octree from one partition to the next. The iteration starts with partition Π_0 of the original mesh $M = M_0$ and works as follows.

Given a partition Π_i and its mesh M_i, the subsequent partition Π_{i+1} is found by rotating M_i whenever i is odd and constructing the partition of the rotated mesh as described in § 6.1. When i is even, the partition is constructed with the non-rotated mesh.

Next, the cuts that describe the DAG between partitions Π_i and Π_{i+1} are computed by looking at how the tetrahedra of the floor segments S_i^j spread over the ceiling segments S_{i+1}^k. The segments of partition Π_{i+1} are simplified (leaving their borders unchanged) which results in the approximated mesh M_{i+1} of the mesh M_i.

The rotation is chosen to include as many borders of the segments into the segments of the consecutive partition. We use a heuristic that is experimentally varified to include many segment-boundaries of a partition into the segments of a consecutive partition. The mesh is rotated by an angle of 45 degrees around the axis $\mathbf{a}$ which is chosen such that the mesh is rotated around the two

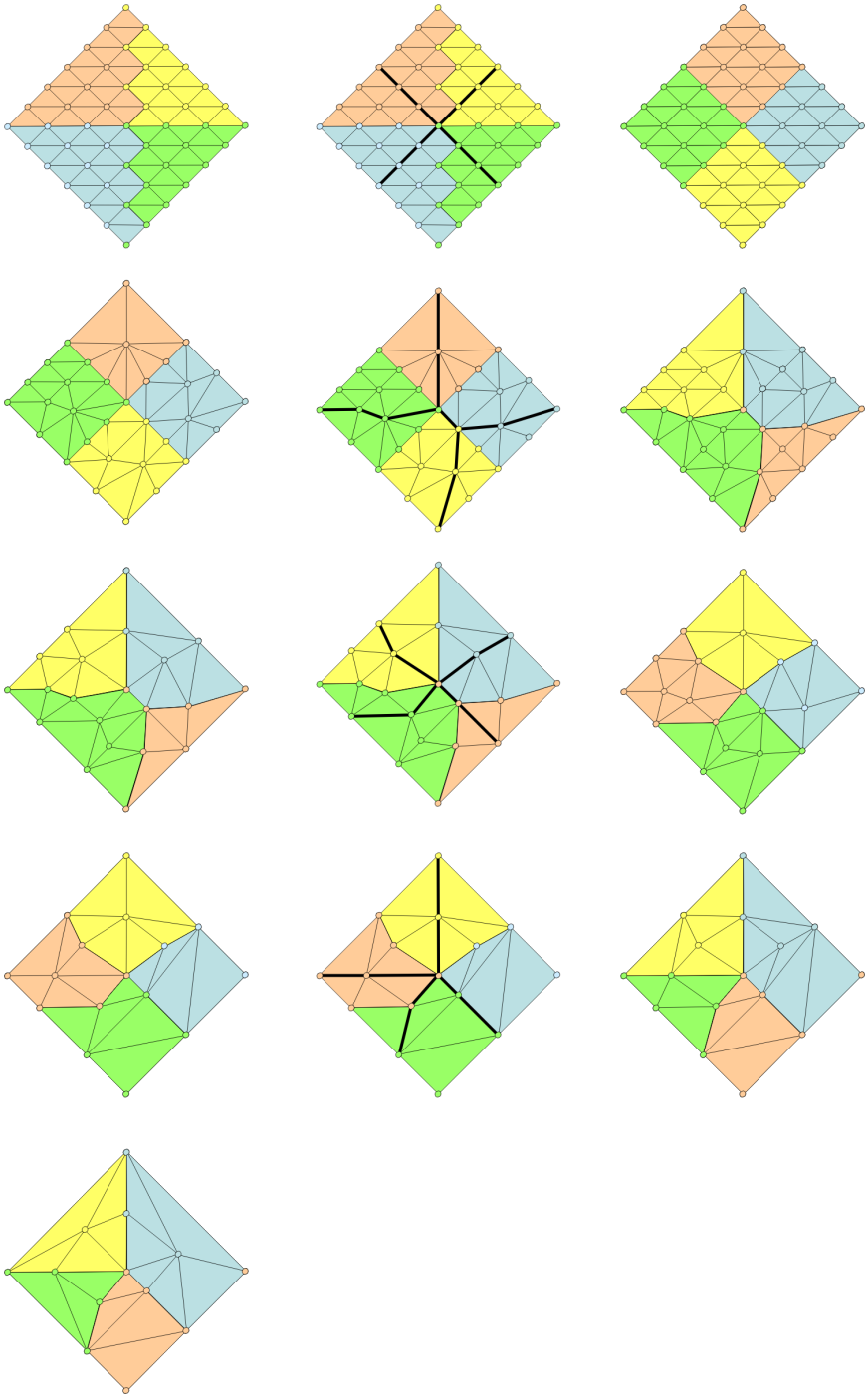


Figure 6.17: A 2D example. Each row shows a partition (left), how this partition is cut (middle) into the consecutive partition (right) which is simplified (next row, left). The process terminates if the mesh is simplified down to a given complexity.

shortest axis of its bounding box as shown in figure 6.16. The center of rotation is the centroid of all vertices.

Note that the fixed segment–boundaries are the only restriction of our simplifier. The simplifier is not overrestricted or needs to decide explicitely when the segment–boundaries shall be simplified. Based on the selection of the partitions, the border elements that are left unchanged in the simplification of one partition are simplified *automatically* in the consecutive partition.

It is important that the leaves of the octree are merged. Otherwise (if, for instance, a simple $\sqrt{2}$-subdivision would be used), the segments as well as the patches are highly un-balanced and would produce a large number of patches with few vertices and tetrahedra. A large number of patches would result in a DAG with many dependencies which would ruin the efficiency of the multi-triangulation.

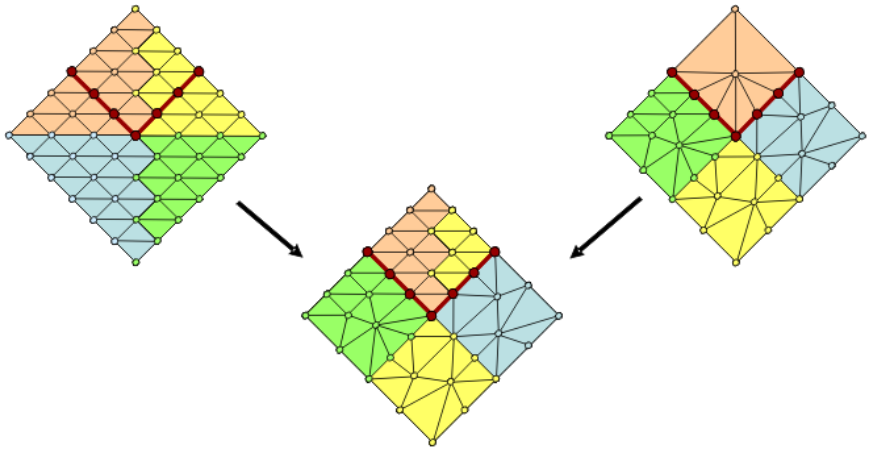


Figure 6.18: A situation of the 2D example of figure 6.17. The red edges have been left unchanged such that the simplified segment at the top can be replaced by patches of two higher–resolution segments.

The index assignment of a segment is chosen to balance the size of the segments. Because a tetrahedron is added to the lowest segment of all its four vertices, we enumerate the segments in the order of their included number of vertices. The segment that contains the most vertices gets the highest index while the segment with the fewest vertices gets the lowest index.

6.3.3 Compression

We store the segments one after the other in a single binary file. For every segment, we store its patches in any order. Because every patch stores its tetrahedral mesh, the file size can become large and we implemented a compression scheme that encodes the meshes of the patches and still supports the multi-triangulation with all its coarsening and refinement operations.

The cut-border machine [GGS99] is a well-known compression scheme for tetrahedral (and triangle) meshes that achieves high compression ratios and performs very quickly. But the cut-border

machine treats the tetrahedral mesh as a non-segmented mesh and cannot restore adjacency information between segments as it is needed by the multi-triangulation. Additionally, it changes the order of the vertices during encoding, i.e. each vertex gets a new index. So, a pre-computed mesh layout is destroyed and replaced by the mesh layout that the encoder introduces. Recent works [YLPM05] have shown that a well-computed mesh layout can drastically improve the rendering performance.

We extend the cut-border machine to support cross-segment adjacencies and to respect any mesh-layout. We will shortly describe the principles of the cut-border machine for tetrahedral meshes followed by a description of our extensions.

Basic Cut-Border Machine

The cut-border machine (CBM) is a region-growing compression scheme which starts at an arbitrary tetrahedron. The border of the region is called cut-border and divides the mesh into an inner part and an outer part which contain the already encoded tetrahedra and the still-to-be-encoded tetrahedra, respectively. The cut-border is a (sometimes non-manifold) triangle mesh that consists of faces of the tetrahedra. Tetrahedra are added to the inner part at a specific triangle face which is called the gate. After a tetrahedron has been added to the inner part, the cut-border is adapted to include the new tetrahedron and the gate is relocated to another face on the cut-border. The CBM stops when the outer part is empty, i.e. when all tetrahedra are encoded.

When a tetrahedron is added to the inner part, three of its vertices are determined by the gate while one vertex is free. Three basic cases describe how this *free vertex* can be located with respect to the gate: (1) the gate can be a border (there is no tetrahedron and thus no free vertex), (2) the free vertex is already on the cut-border, or (3) the free vertex is not on the cut-border and is hence a new vertex. Basically, the encoder determines which of the three cases applies to the current tetrahedron at the gate and encodes the case as a symbol. In order to achieve good compression ratios, case (2) is subdivided into several sub-cases as shown in figure 6.19: Flip, Top, Close, and Join. The mesh is traversed and tetrahedra are added to the inner region which creates a stream of symbols that is arithmetically encoded. The geometry of the vertices may be interleaved with the stream of symbols or may be encoded in a single block. So, the vertices are encoded in the order of their appearance due to the traversal order. The traversal order is chosen such that the vertices are inserted into a fifo buffer in the order of their appearance starting with the four vertices of the first tetrahedron. The first vertex of the fifo is fixed and all its incident tetrahedra are traversed and encoded (or decoded). Then, the next vertex of the fifo is fixed and processed.

Each triangle of the cut-border mesh stores its three edge-neighbors. As long as the cut-border is manifold, this neighboring information can be easily updated during the traversal. But the sub-cases Flip and Join can introduce non-manifold edges which need to be detected. The edge-neighbors must be set correctly even for non-manifold edges. [GGS99] propose a data structure that stores for each vertex of the cut-border a set of all its incident triangles. To detect a non-manifold edge, the sets of both end vertices of the edge are cut. If the cut is not empty, the edge is

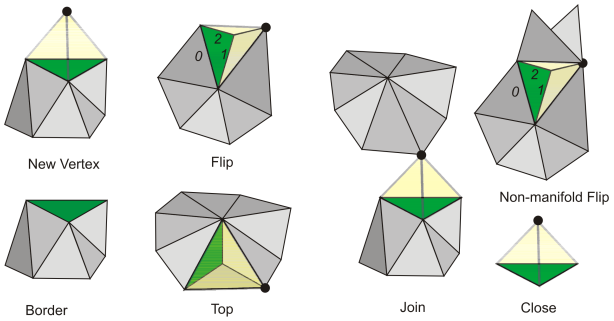


Figure 6.19: All possibilities of how a vertex can be located to the gate. The new tetrahedron is shown in transparent light gray whereas the gate is very dark gray. The new vertex (black circle) together with the gate forms the new tetrahedron. The Flip is a $Flip_1$ operation because the new tetrahedron is created by (something like) a flip of edge 1 *of the gate.*

already in the cut-border and the edge-neighbors of the newly added triangles are adapted correctly using either the geometry information (if known) or the (possibly) encoded triangle indices.

Cut-Border Machine for Multi Resolution Meshes

Our multi triangulation consists of segments each of which has several patches. We encode the segments one after another and for each segment we encode its patches one after another. The basic entity that we compress is the tetrahedral mesh of a patch.

Basically, the vertices are enumerated segment-wise. Vertices that lie on segment boundaries (i.e. tetrahedra of different segments reference the vertex) are copied in each segment. So, the tetrahedra of a segment always reference points whose geometry is also stored in the segment. For every such border vertex, an additional index is stored that is unique for all segments of the multi-resolution mesh. So, a vertex can be identified uniquely although it might have several copies in different segments.

We start with an arbitrary tetrahedron of a patch and grow the region that is bounded by the cut-border. Every time when a tetrahedron is added to the inner part, an *opcode* describes the position relative to the cut-border. We have a total of 11 opcodes that fully determine all possible positions.

- *Border* (B): the gate is a border of the (whole) tetrahedral mesh.
- The free vertex has not been visited yet.
  - *New Shared Vertex* (S): The free vertex lies on a border between patches or segments (i.e. there are tetrahedra of different segments or patches that share this vertex).
  - *New Vertex* (N): The free vertex is not on a border between segments or patches.
- The free vertex has been visited yet.

- *Flip*$_i$ (F_i): The free vertex is incident to exactly one adjacent triangle of the gate. i determines which of the three adjacent triangles has the free vertex in common.
- *Top*$_i$ (T_i): The free vertex is incident to exactly two adjacent triangles of the gate. i determines which of the three adjacent triangles is not incident to the free vertex.
- *Close* (C): The free vertex is incident to all three adjacent triangles of the gate.
- *Join* (J): The free vertex is incident to none of the adjacent triangles of the gate.

The N, S, and J operations get the index of the new vertex as parameter. The index of the new vertex can be encoded with at least $\lceil \log(n) \rceil$ bits where n is the number of vertices of the segment. For performance reasons, we encode the index always with 16 bits. Note that T_i and F_i do not need this parameter because the free vertex is located at an edge-incident triangle of the gate which is fully determined by the index i. The S operation needs an additional parameter which describes the unique index of this shared vertex and is encoded with additional 32 bits (theoretically, we would need at most $\lceil \log(m) \rceil$ bits where m is the number of vertices of whole multi resolution mesh). As example, the connectivity of a cube that is subdivided into 6 tetrahedra could be represented as 0236BBBN1BBN4BN7BN5BBTBB. The first four vertices define the first tetrahedron. We store the opcodes as 4-bit-words and did not implement arithemtic coding because the reduction of the file sizes was sufficient for our purposes. But of course the compression ratios could be additionally improved by an arithmetic coder.

In contrast to [GGS99] we detect non-manifold edges with a hash map that stores for every edge of the cut border all its incident triangles. An edge is uniquely defined by both its end vertex indices. So, we avoid the cut of two sets of triangles and can immediatly lookup which triangles are incident to an edge. We only allow non-manifold edges that are incident to four triangles. If the encoder detects a non-manifold edge that is incident to more than four triangles, it outputs a *Skip* symbol (increasing the number of symbols to 12), discards the current gate and proceeds to the next gate without doeing any action. Note that if four triangles are incident to an edge, the adjacency information is uniquely defined by the order of the vertices in the triangles and no geometry tests must be performed which easies the implementation a lot but decreases the compression ratio.

The traversal order is crucial for the efficiency of the algorithm. We follow [GGS99] and traverse the mesh point-wise starting with the four points of the first tetrahedron.

The decoder traverses the stream and constructs the mesh based on the arrival of the opcodes. Because we distinguish between in-patch vertices (opcode N) and shared vertices (opcode S) we can reconstruct each of the four face-incidencies of a decoded tetrahedron as follows. If all three vertices of a face are shared vertices, it must be a cross-patch (or cross-segment) face. Otherwise, the face is an in-patch face. Two hash maps store pairs of triangles which represent the face-incidencies. One hash map handles in-patch face-incidencies while the other hash map handles cross-patch incidencies. An in-patch face registers to its in-patch hash map. If there is no triangle registered yet, it stores its own face incidency information. If there is a triangle already, the face-incidency information is updated for both the stored face and the own face and the hash entry can

be deleted.

A cross-patch face looks up the cross-patch hash map with the triangle that is formed by the unique indices. If there is no triangle registered yet, the face inserts itself into the hash map. If there is just one face registered yet, the face updates its incidency using the stored face and stores itself as second face into the hash map. If there are already two faces stored, the patch must have replaced a different patch by a coarsening or refinement operation. This other patch must either consist to the ceiling or floor segment. So, this patch can be identified and its according face in the hash map which can be replaced with the current face.

The geometry and attribute values are stored in a single table per segment. Note that our scheme also supports a compression that does not respect the mesh layout. The opcode N would get no parameter and the vertices are enumerated in the order of arrival. The incidency information could be restored by the unique index of opcode S.

6.3.4 View-dependent Rendering

We integrated the multi triangulation into a visualization system that supports interactive direct volume renderings, isosurface extraction and vector field visualization. We first describe how the mesh can be adapted followed by the particular visualization techniques.

Updating the cut

As described in § 6.3.1, a consistent mesh is represented by a graph cut through the DAG. Moving this cut down refines the mesh while moving it up coarsens the mesh. The cut may not be moved freely but needs to fulfill the conditions for a consistent mesh as shown in figure 6.14.

The edges of the cut represent patches, so we represent the cut as a list of active patches. A patch is refined by activating all outgoing patches of its floor segment whereby the graph cut moves down and the patch itself is deactivated. But in order to get a consistent mesh, every path from the roots to the leaves must contain exactly one active patch. So, a patch can be refined only if all its ceiling segments do not have active patches that precede the ceiling segment on the way from the roots to the segment. If there are such active preceding patches, they must be forced to refine. Otherwise, patches would overlap and the mesh is not consistent.

A patch can be coarsened and replaced by all incoming patches of its ceiling segment if all its siblings (i.e. all outgoing patches of the ceiling segment) are active. If they are not, the patch cannot be coarsened.

Two queues maintain the cut, a coarsening queue and a refinement queue similar to [CDFM+04] and § 5.2. The refinement queue contains all patches that are active and can be refined by activating all patches of their floor segments. The coarsening queue contains all patches that can be activated because all of their siblings are active. After each refinement or coarsening, the queues are updated. The queues are sorted by an error value that is precomputed and stored with each segment. The kind of the error value depends on the rendering technique and is described in the following sections.

We first try to coarsen as many patches as possible and refine afterwards as many patches as necessary. The user can specify the number of tetrahedra that the adapted mesh can maximally contain. A refinement is performed only if the so-refined mesh does not contain more tetrahedra than the given maximal number (including the tetrahedra that are created by forced refinements).

We choose a caching scheme to keep track of the patches in the core memory. First of all, only complete segments are loaded or discarded. Second, a segment is discarded only if all its incoming and outgoing patches are deactive. So an active neighborhood surrounds always the current graph cut and provides all segments that might be needed by future frames.

Direct Volume Rendering

We normalize all scalar attribute values of the mesh to the unit interval $[0, 1]$. Every segment stores an error histogram H for every attribute channel. H contains for each of its attribute slots the largest error within the segment, measured to the original tetrahedral mesh. These errors increase from the leaves of the DAG to the roots.

The attribute error of a segment is weighted by the (user-chosen) classification. Assuming that the classified opacity is stored in a lookup-table C of a given resolution N and that the error histograms have the same resolution, the attribute error E_a can be computed by the simple scalar product

$$E_a = V \sum_{i=1..N} H(n)C(n).$$

So, every error value $H(n)$ is weighted with its classified opacity $C(n)$ which ranges from 0 (not visible) to 1 (fully visible). Attribute values that are classified as visible and that have a large error, will strongly contribute to E_a. Otherwise, attribute values that are classified as not visible will not contribute to E_a even if their attribute error is large. The V term is zero if the segment is outside of the view frustum and one otherwise. So, segments that are not visible have a zero error and will be coarsened. The segments in the queues are sorted by the attribute error E_a.

The direct volume renderer uses projected tetrahedra with a preintegration technique. Note that the normalization of the attribute values is necessary for the preintegration technique and for a quadric-based simplificiation algorithm during the construction of the mult-triangulation.

Isosurfaces

Isosurfaces can easily be computed because the tetrahedral mesh is consistent. We implemented the extraction of isosurfaces on the CPU as well as on the GPU. Regardless which algorithm is chosen to implement the isosurface extraction, the algorithm can perform on a per-patch basis. The isosurface can be extracted from each patch independently of neighboring patches. The multi triangulation ensures that the border of the patches are connected consistently and hence the isosurfaces of different patches match perfectly at the patch borders.

In addition to the attribute error of the direct volume renderer, every segment stores a histogram of spatial errors for all attribute channels. Using these histograms, the mesh can be refined at those

areas where the isosurface passes through. The renderer supports both opaque and transparent isosurface rendering. The transparency as well as the color of an isosurface can be chosen by the user.

Flow Visualization

Stream lines for flow visualization can be extracted without any effort because the mesh is consistent and maintains a single view to the mesh. The algorithm of section § 7.3 is fully applicable.

6.3.5 Results

We tested the multi triangulation with the datasets listed in table 6.7. The implementation runs on a standard PC with Pentium 5, 1 GB main memory, a GeForce FX graphics card and Windows XP.

The preprocessing step includes the partition of a tetrahedral mesh into segments using an octree, simplifying the segments and storing them onto disc. The timings and the size of the resulting files are given in table 6.7 and include file I/O.

The properties of the created segments are also shown in table 6.7. The average size of the segments is reported as well as the average size of the patches, i.e. the cut between segments. The tetrahedral simplifier uses quadric error metrics, see §§ 4.1,3.1.4 and runs stable with vertex distribution quadrics and feature edges.

The performance of the decoder is neglectible. We load the coded stream of a segment into the main memory and can decode about 500K tetrahedra per second.

Depending on the users choice, the mesh gets adapted to an error or to a maximal number of tetrahedra. We examined that a maximal number of tetrahedra is mandatory to be defined because otherwise too many tetrahedra are created for a low error threshold. This is due to the fact that our system currently does not take occlusion by the segments into account. Only the attribute error is considered. Figures 6.20–6.23 show renderings with the visualization system.

Nevertheless, the system runs with a performance that is mainly bound by the direct volume renderer which currently in our implementation has a peek performance of about 800.000 tetrahedra per second and supports pre-integration in the pixel and fragment shaders. The isosurfaces are extracted as triangle meshes and rendered using the standard triangle graphics pipeline.

Some regions of the tetrahedral mesh need more tetrahedra to be created than the given threshold in order to fulfill the error tolerance. This is caused by the dependencies between the segments. For these regions the user has to increase the number of maximal tetrahedra and needs to take a drop of the fps into account.

The operating system supports our caching scheme. At the beginning, the first segments that are to be displayed need to be loaded. As soon as this has been finished, the loading of segments doesn't drop the fps rate of the renderer. Additional, one has to watch that the file is not defragmented on disc which causes the header of the disc to search for parts of the file and causes the system to slow down.

Name	Sea	Fighter	Rbl	F16	Earthquake
# Vertices	102,165	256,614	730,273	1,124,648	2,461,694
# Tetra	524,640	1,403,504	3,886,728	6,345,709	13,980,162
# Tetra in base mesh	20K	30K	110K	80K	120K
# Segments	66	181	479	798	1,645
# Avg Tets per segment	15,694	15,873	16,956	16,676	20,416
# Patches	217	1,207	2,754	6,991	12,857
# Avg Tets per patch	3,749	2,369	2,906	1,830	2,604
Preproc. Time [hh:mm:ss]	0:10:10	0:20:15	0:39:26	0:40:31	1:12:45
File size uncompr. [MB]	41	100	285	565	1,200
File size compr. [MB]	12	21	70	102	265

Table 6.7: The properties of the datasets and the timings for the construction of the multi triangulation from the original mesh. The segmenting octree was steered to contain at most 5,000 vertices per leaf node.

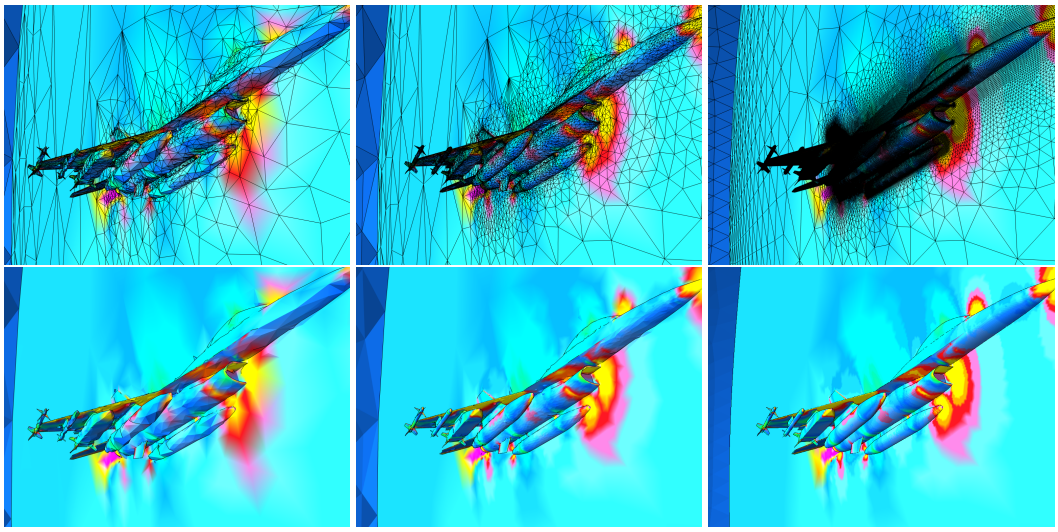


Figure 6.20: The F16 contains about 6 million tetrahedra. Left: lowest resolution, about 80K tetrahedra. Middle: adapted resolution, about 200K tetrahedra. Right: full resolution, 6 Mio tetrahedra. Only the boundary faces are drawn.

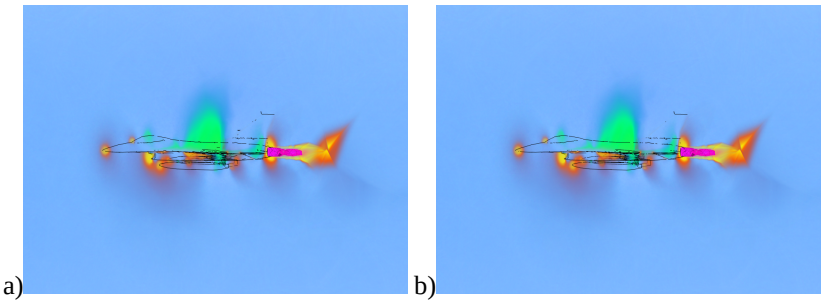


Figure 6.21: A direct volume rendering of the F16. (a) Adapted mesh with about 300K tetrahedra, (b) the full mesh with 6 Mio tetrahedra.

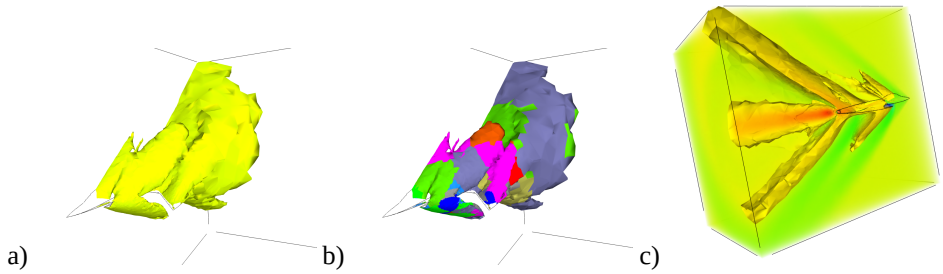


Figure 6.22: The isosurface at isovalue 0.2 of the Fighter model. (a) Calculated from an adapted mesh that contains 120K tetrahedra. (b) The isosurface of (a) consists of several parts which are rendered with different colors. Each part is calculated from a patch of the multi-triangulation. (c) The isosurface rendering is embedded into a (coarse) direct volume rendering.

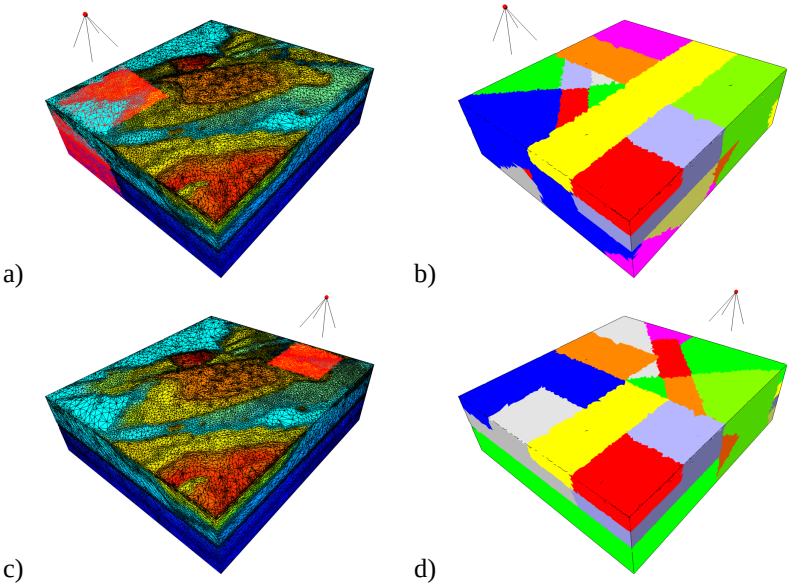


Figure 6.23: The Earthquake dataset observed at different positions.

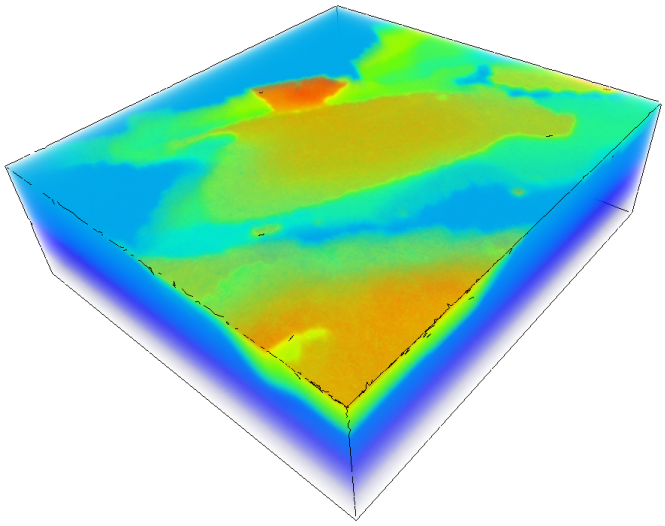


Figure 6.24: An interactive direct volume rendering of the Earthquake dataset with a workload of about 400K tetrahedra. The original dataset contains more than 13 million tetrahedra and cannot be visualized interactively by a direct volume rendering with current graphics hardware.

Chapter 7

Visualization

The words or the language, as they are written or spoken, do not seem to play any role in my mechanisms of thought. The psychical entities which seem to serve as elements in thought are certain signs and more or less clear images which can be "voluntarily" reproduced and combined. - *Albert Einstein*

The visualization of 3D scalar, vector or tensor fields is challenging because the large amount of information will either be mixed during projection from 3D to 2D or lead to severe occlusion problems. This chapter introduces three novel techniques for exploring huge scientific volume datasets.

Traditionally, scalar fields can be visualized *directly* by direct volume rendering or *indirectly* by isosurface extraction and rendering. While isosurface extraction usually leads to huge triangle meshes that can be visualized by a standard graphics pipeline, direct volume renderings rely on an optical model describing how light interacts with the scalar field inside the volume.

The interaction of light with a volume can be described by a particle model that the transport equation is applied to [Max95]. Typically, absorption and emission of light are considered while scattering is suppressed. The model is applied to rays following the light flow from the volume to an observer. The ray is parameterized with a length parameter s such that $r(s)$ specifies a position in $\mathbf{R}^3$. Absorption is given by an optical density function $\tau(s)$ which describes the rate of occlusion at $r(s)$, i.e. the fraction of light occluded by the volume at position $r(s)$. A generating term $g(s) = \kappa(s)\tau(s)$ describes the emission of light at position $r(s)$. Combining both emission and absorption, the interaction of light with the volume along a ray is described by the differential equation:

$$\frac{dI}{ds} = g(s) - \tau(s)I(s) = \kappa(s)\tau(s) - \tau(s)I(s) \tag{7.1}$$

where $I(s)$ is the light intensity at position $r(s)$. The optical properties $\tau(s)$ and $\kappa(s)$ are evaluated at certain positions $r(s)$. Traditionally, a *transfer function* maps a scalar field value f at a given position $r(s)$ in the volume to an optical property.

Following [Max95], the differential equation (7.1) can be solved by the *volume rendering integral*

$$I(u) = I_0 \exp\left(-\int_0^u \tau(t)dt\right) + \int_0^u g(s) \exp\left(-\int_s^u \tau(t)dt\right) ds \tag{7.2}$$

with I_0 as the light intensity at distance $s = 0$ where the ray enters the volume.

For an efficient numerical integration, the integral (7.2) is approximated by Riemannian sums that assume constant volume properties within each Riemannian integration interval. In order to enable a fast computation of the approximating sum, the volume must be traversed from front to back or back to front introducing a *visibility sorting* which can be done easily for regular volume grids [Ake93, GWGS02] but is not trivial for unstructured grids like tetrahedral meshes. Starting with the MPVO algorithm of [Wil92] which basically constructs and traverses an adjacency graph of directed edges and works for all meshes with cycle-free such graphs, a lot of subsequent work has been done including XMPVO [SMW98], BSP-XMPVO [CKM+99], MPVOC for cyclic meshes [KE01] or the k-buffer for hardware-assisted sorting [CICS05].

Direct Volume Rendering for tetrahedral meshes approximates the volume rendering integral by sorting all tetrahedra from back to front and projecting them onto the 2D image plane. The four vertices of a tetrahedron span at most four triangles in the image plane which are rendered and blended in the framebuffer with previously projected tetrahedra. The seminal work of [ST90] has introduced *projected tetrahedra* and has later been improved by preintegration techniques [GRS+02, SCCB05, CBPS06] to elevate the picture quality and ensure correctness under perspective projections [KQE04]. They typically precompute the volume rendering integral for different front and back colors as well as thicknesses of tetrahedra. The precomputed integral values are stored in look-up tables which can be easily accessed at run time from texture memory.

Indirect volume renderings extract isosurfaces from the volume which are triangualar meshes. The marching tetrahedra algorithm is simple to implement in both software and graphics hardware [Pas04, KSE04, KW05]. Basically, a marching tetrahedra algorithm checks for each tetrahedron if it is passed by an isosurface. Assuming linear interpolation inside a single tetrahedron, the intersection between an isosurface and a tetrahedron results in at most two triangles depending on how many edges of the tetrahedron are cut by the isosurface.

For large tetrahedral meshes with many very small tetrahedra, the isosurface extraction algorithms usually leed to triangular meshes with hundreds of thousands of triangles. These triangle meshes can be rendered using graphics hardware. Often, the user can add transparency to each isosurface in order to increase perception. But this introduces an additional sorting step which can become time–consuming for huge triangulated isosurfaces. § 7.2 discusses a rendering technique that samples the isosurfaces with discrete points and renders point clouds only which naturally allow for a transparency without sorting due to their discrete nature.

7.1 Silhouettes

Direct rendering approaches for unstructured tetrahedral meshes often fail to inform the user about its current position and orientation within the dataset. Additional features like bounding boxes or glyphs help to improve the orientation. I introduced a rendering technique for silhouettes which is simple to implement and is based on the detection of sharp edges of the tetrahedral mesh [SS05a].

Conceptually, the boundary of a tetrahedral mesh is a triangular mesh. Each triangle has a normal associated with it which points outside the tetrahedral mesh. For each edge of the triangular mesh, the normals $\mathbf{n}_1$ and $\mathbf{n}_2$ of both its adjacent triangles are compared to the viewing direction $\mathbf{v}$. An edge is a silhouette edge if and only if the dot products between the normals and the viewing direction $\mathbf{v}$ have different signs, i.e. $\langle \mathbf{n}_1, \mathbf{v} \rangle \times \langle \mathbf{n}_2, \mathbf{v} \rangle < 0$.

Instead of computing and storing the boundary triangular mesh, the algorithm operates directly on the tetrahedral mesh and a caching scheme. This enables the full support of a multi-resolution model where the cache needs to be updated every time the mesh is changed – but not otherwise. For instance, if a segment replaces another segment, the entries of the deleted segments must be removed while the entries of the appearing segment must be added.

The cache itself maintains edges each of which consists of the indices of both extreme vertices v_1 and v_2 and two normals $\mathbf{n}_1$ and $\mathbf{n}_2$.

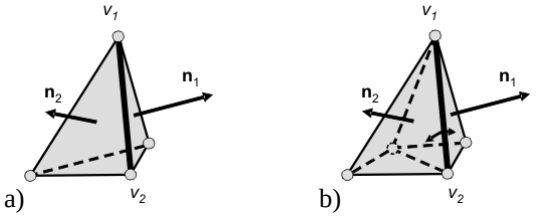


Figure 7.1: A silhouette edge maintains its two extreme vertices and the normals of both its adjacent boundary triangles which may be found by a simple mesh traversal as shown in (b).

A single run over the tetrahedral mesh initializes the cache as follows. If a tetrahedron has two boundary faces, the edge that is adjacent to both boundary faces is added to the cache together with the according face normals. Remember that all tetrahedra have an orientation such that their volumes are positive and thus the normals point outwards. If a tetrahedron has just a single boundary face f, the (at most three) tetrahedra are found whose boundary faces are adjacent to f. This can be done efficiently looping around all three edges of f. Again, each such edge together with the normals of its adjacant boundary faces is added to the cache.

In contrast to many silhouette techniques for triangular meshes, hidden silhouettes are not removed to the advantage that a direct volume rendering can contain transparent parts and silhouette edges of backfacing boundary mesh parts are still visible.

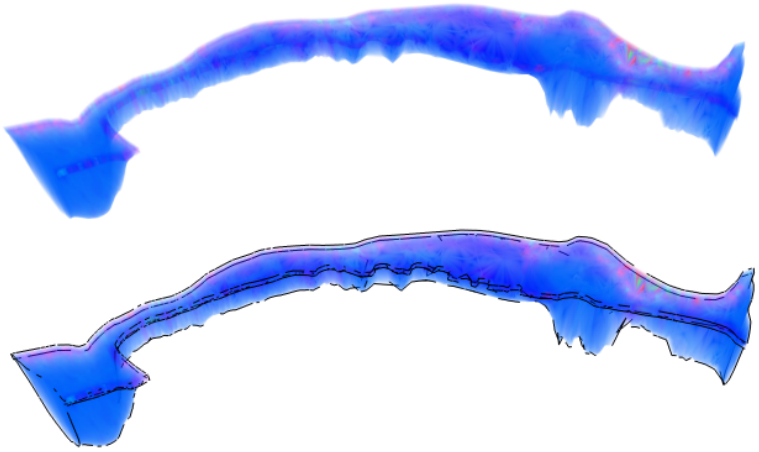


Figure 7.2: The Sea Dataset without and with silhouettes.

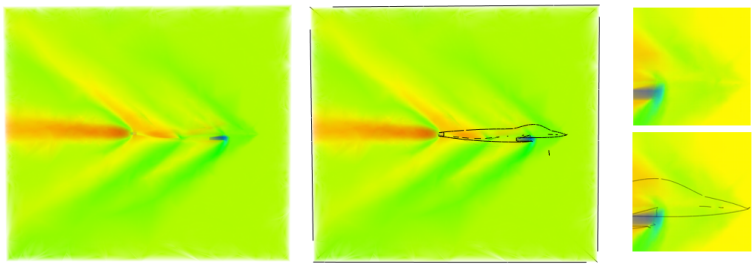


Figure 7.3: The Fighter Dataset without and with silhouettes. Right: A zoom into the dataset.

7.2 Point Rendering for Scalar Fields

Isosurfaces are an important tool to understand structural properties within scalar fields and are heavily used by scientists to explore datasets. But if multiple isosurfaces shall be visualized, they may occlude each other making looks into inner regions of the dataset impossible. Transparent isosurfaces (where the user can choose the rate of transparency for each isosurface) is a widely used solution for the occlusion problem but introduces a depth-sorting of all isosurface triangles for each frame.

The technique proposed by this section is solely based on point rendering in order to visualize unstructured datasets [SS05b]. As the basic idea, points are distributed over the volume domain at specific important locations and are colored and lit with a lighting model adapted to the scalar field. Due to their discrete nature, point renderings allow the user to look inside inner regions of a dataset *without* any sorting. The distribution of points enables even non-manifold meshes to be visualized easily as well as meshes that contain tetrahedral cycles or non-convex regions such that a correct depth-sorting becomes complex.

I assume that all attribute values v are in a given attribute interval $I = [v_{min}, v_{max}]$, i.e. $v \in I$.

The attribute values do not need to be normalized (as they would be for direct volume rendering with pre-integration).

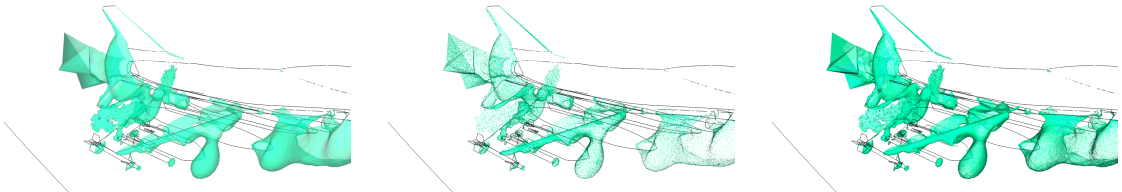
Conceptually, a dense set of isosurfaces is extracted uppon which points are distributed in a way that the distances between points on one isosurface are lower than the distance of points between different isosurfaces.

In order to create a dense set of isosurfaces, a user-given number n of isovalues v_k is chosen from the interval I. In the simplest case, the isovalues are regular-spaced numbers $v_k = v_{min} + k\frac{v_{max}-v_{min}}{n}$. Next, the tetrahedral mesh is traversed and each tetrahedron is tested if it contains any of the isovalues v_k, that is, each tetrahedron is assigned triangles of isosurfaces at values v_k. There may be none, one, or two triangles for each tetrahedron per isovalue v_k.

Each triangle is point sampled with the goal to distribute the points uniformly across the whole isosurface. Therefore, the user provides a threshold A which specifies the area of an isosurface that shall be sampled with a single point. If the area of a triangle is larger than A, it is subdivided until the areas of the subdivision triangles are below A. If the area of a triangle is smaller than T, a random selection happens where the randomness depends on the size of the triangle relative to A. A dice D is evaluated and a point is rendered if

$$D > \frac{|t|}{A}.$$

Because D is evaluated only for $|t| < A$, the ratio is bound between 0 and 1, i.e. $0 \leq \frac{|t|}{A} \leq 1$. This upsampling is a very important feature and the main difference to transparent isosurfaces with triangles because a triangle mesh cannot be simplified with such a simple technique. Often, the isosurface contains many more triangles than needed for a given viewport.



a) b) c)

Figure 7.4: The isosurface at isovalue=0.35 of the F16 model contains 350K triangles and is rendered semi-transparent (a). An upsampled point cloud with 90K point (b) and without upsampling and 407K points (c).

The random process is simulated by a fixed array of random variables that must be produced once. Currently, the array contains 100 such random variables ranging from 0 to 1. Using such an array might bias the randomness but is much faster to evaluate than a function which generates random variables. The results of visualization, however, are not affected noticeably.

Depth scaling can be applied in order to reduce the number of points per area that are far away from the observer. Again, a randomized algorithm throws a dice D_d which is for each point

compared against a distance threshold

$$D_d > \max\left(0.1, \min\left(0.9, \frac{d}{d_{max}}\right)\right)$$

where d is the distance of the point from the observer and d_{max} is the maximal such distance, usually set to the extent of the bounding box. The min and max functions restrict the threshold to lie in the unit interval. For an elevated interactivity, the points can be sorted into a simple grid and d is evaluated once for each grid cell instead of each point.

The points are lit using the triangle's normal of the isosurface. Although the gradient could be used, the numerical estimation of its direction is not as robust as the estimation of triangle normals which can be taken directly from the isosurface. Because triangle normals are uniquely defined except for their sign, the normals must be oriented to show into the direction of an observer. If the scalar product between the viewing direction and the normal is positive, the normal points away from the observer and must be reversed.

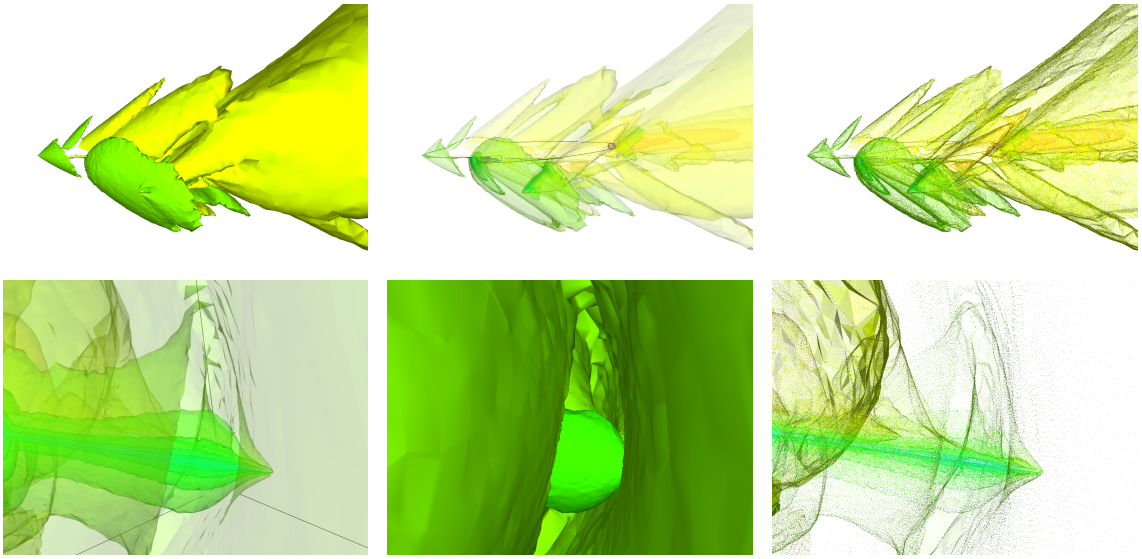


Figure 7.5: The Fighter Dataset with opaque isosurfaces ($>$ 1 million triangles), semi-transparent isosurfaces and point rendering (500K points). Second row: A zoom into details at the wing area.

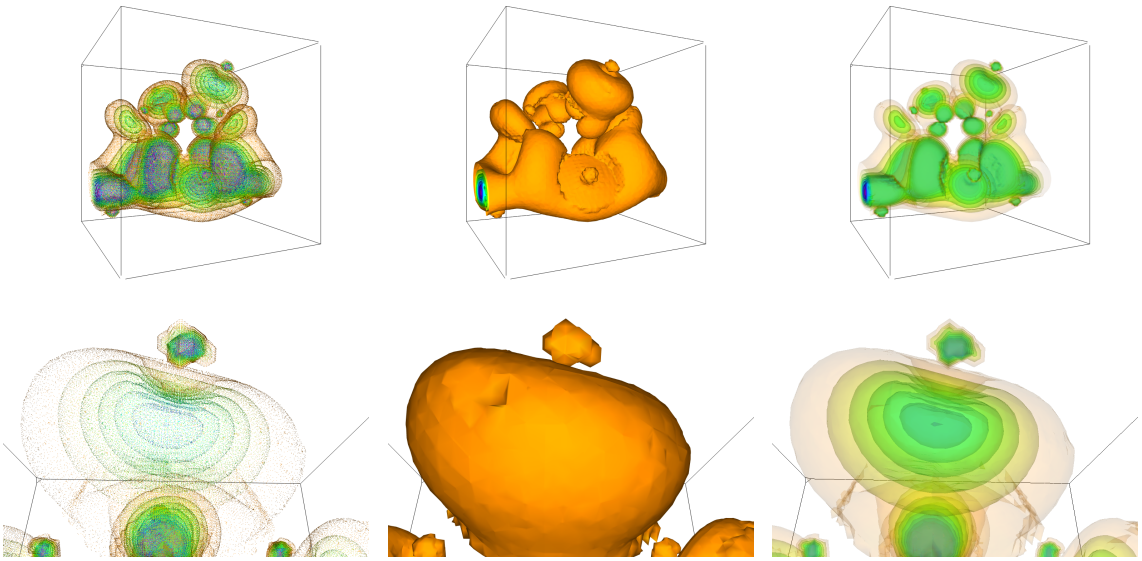


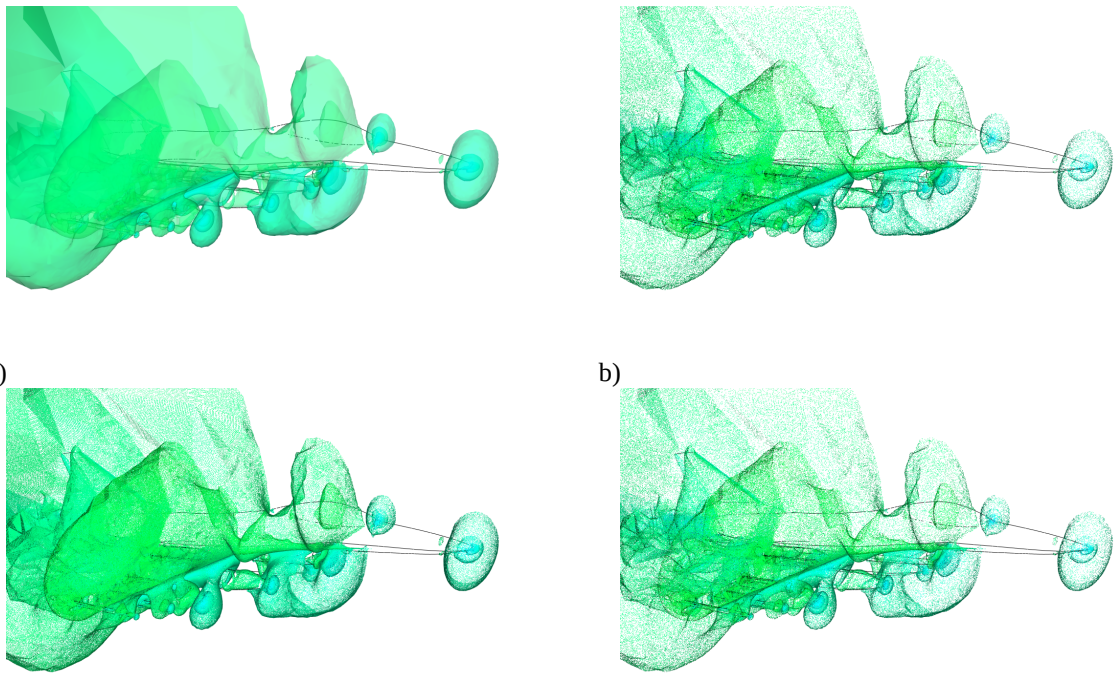
Figure 7.6: The Neghip Dataset with point rendering, opaque isosurfaces and transparent isosurfaces. Bottom row: a zoom into the dataset.

7.3 Stream Lines for Vector Fields

The visualization of vector fields differs from the visualization of scalar fields in the directional information which must be clearly communicated. A frequently used visualization technique traces stream lines or path lines through the domain which can be rendered as line strips. A stream line visualizes the path of the flow but not the direction itself of the flow which can be either forward or backward along the stream line. Additional icons like arrows must be drawn in order to illustrate the flow direction. Several algorithms exist to place these stream lines uniformly and visually appealing or around regions of particular interest like sources or sinks of the vector field. Nevertheless, the rendering of stream lines for threedimensional vector fields results in heavy visual cluttering caused by stream lines that occlude other stream lines during projection from 3D to 2D.

As an alternative rendering technique that exploits existing graphics hardware, texture advection [WE04] can be used to visualize vector fields. Thereby, a noise texture is blurred into the direction of the flow similar to line integral convolution (LIC). If additional transparency information is carried by the vector field through the texture, looks into the inside of a dataset become possible. The major drawback of texture advection techniques is their restriction to regular grids and their discrete resolution which makes it difficult to capture important regions of the vector field. Nevertheless, they visualize the vector field continuously without gaps as they naturally appear by discrete stream lines.

In [SS06b], a visualization technique for vector fields has been proposed which integrates a dense set of stream lines in a vector field over a tetrahedral mesh and enables the user at visualization time to select some of these stream lines for opaque rendering while non-selected lines



a) b)

c) d)

Figure 7.7: The isosurfaces 0.2, 0.4, 0.6 and 0.8 of the F16 dataset comprise 1.4 million triangles and are rendered semi-transparent (a). The point sampling contains 400K points and is rendered with decreasing depth correction value d_{max} (b-d).

are hinted in a semi-transparent rendering. In this way, the user can easily look into the most interesting parts of the dataset while all other parts are still hinted.

7.3.1 Integration of Dense Stream Lines

The ultimate goal of stream line placement is to distribute the lines uniformly over a domain of interest, that is, to find a set of lines which cover the whole domain and have similar distances between any pair of lines. While good appoximations exist for 2D vector fields, it is much harder for 3D vector fields.

The approximation presented in this section subdivides the domain of interest into cuboidal cells by a regular grid. The user can specify the resolution and the domain of the grid which is initially suggested to cover the whole domain of the dataset. Stream lines are computed one after another where the integration of a new stream line is started in a grid cell that has not been covered by previously computed stream lines.

Every stream line has an unique id which is stored in each grid cell that intersects the stream line. Beginning with an empty grid, we select a grid cell, choose a seed point from the cell and

integrate a stream line by forward and backward integration of the vector field starting at the seed point. Each integration step uses a second-order Runge Kutta integrator. The id of the integrated stream line is stored in each grid cell. New stream lines are integrated from empty cells, that is, from cells that have not been intersected by stream lines so far. This way, several stream lines may pass a grid cell but new lines are integrated in regions only that have not been visited yet.

During integration, the vector field must be evaluated at various positions $\mathbf{q}$. For this, the tetrahedron which contains $\mathbf{q}$ must be determined. Every such integration step starts at a point $\mathbf{p}$ whose containing tetrahedron t is known (either from the previous integration step or because $\mathbf{p}$ is a seed point). In order to find the tetrahedron containing $\mathbf{q}$, we first check if $\mathbf{q} \in t$. If it is not, we do a tetrahedral walk along the ray starting at $\mathbf{p} \in t$ with direction $\mathbf{v} = \mathbf{q} - \mathbf{p}$. The intersection point of the ray and t is determined which lies on one of the four tetrahedral faces. So, the ray enters an adjacent tetrahedron t_1 which is checked if it contains $\mathbf{q}$. If not, the walk is continued starting at the computed intersection point.

Note that the multi-resolution data structures presented in this thesis fully support tetrahedral walks because the adjacency informations can be queried for any adapted mesh.

In addition, stream lines can be created in the neighborhood of critical points, that is, around points that have a vector of length 0. Therefore, a sphere is centered at every such point and seeds are placed uniformly on the sphere's surface. A uniform distribution of points on a sphere can be achieved by a regular subdivision of the triangle mesh of a tetrahedron.

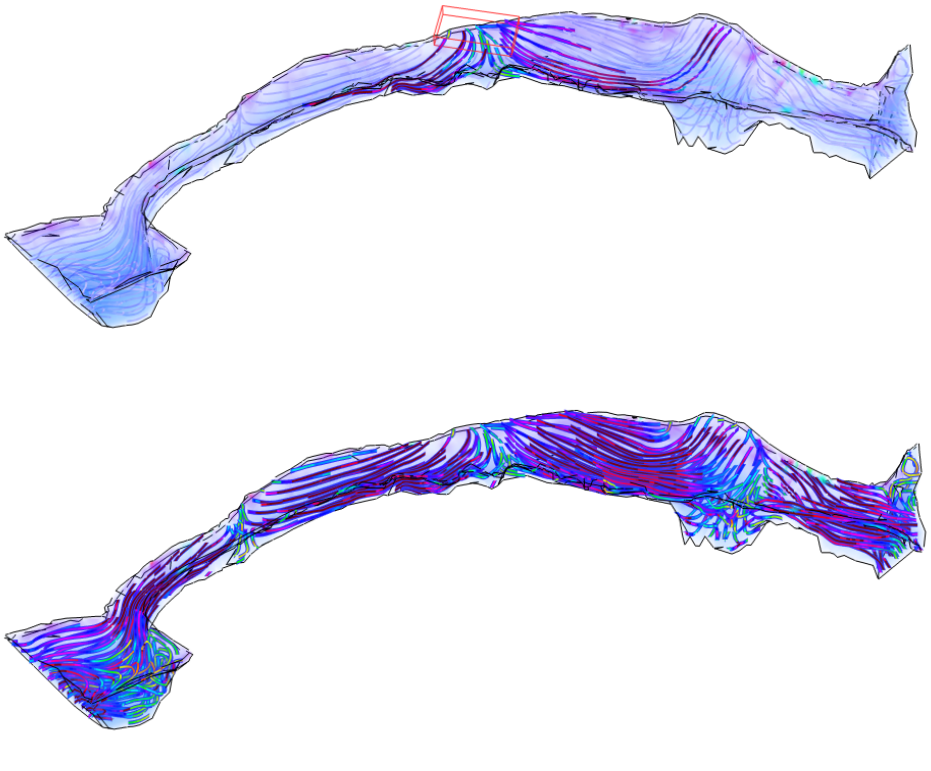
7.3.2 Ghosting

The rendering of all stream lines of the grid produces a lot of visual cluttering and it is hard for the user to perceive spatial relationships between stream lines and to identify global as well as local structures of the vector field. To avoid this, we introduce a technique called ghosting.

The set of stream lines is separated into *full stream lines* and *ghosted stream lines*. Full stream lines are rendered opaque while ghosted stream lines are rendered semi-transparent. The user sees the local and global structure of all full stream lines while he can still observe the overall vector field by the ghost stream lines.

In order to separate the set of stream lines, the user moves a small *selection box* through the domain. All stream lines that pass this box are full stream lines while all other stream lines are ghosted stream lines. Since this technique should be interactive, a fast technique is necessary to identify all stream lines passing the box. For this, the grid data structure is exploited again. All grid cells that overlap with the box can be determined by locating the eight corner points of the box within the grid. All stream lines of every overlapped grid cell are tested for intersection with the selection box. The grid serves as a spatial search data structure.

The rendering process itself first renders all full stream lines with enabled depth buffer. After that, the line segments of all ghosted stream lines are sorted by their distance from the observer and are rendered from back to front with alpha blending and depth buffer enabled. All ghost stream lines are drawn with a single color and transparency adjustable by the user. This rendering



a)

b)

Figure 7.8: An overview with a selection of stream lines and ghosted stream lines (a) and all stream lines (b).

technique ensures high-quality visualizations with a correct depth impression.

7.3.3 Flow Direction

Stream lines do not show the actual direction of the flow which can be either forward or backward along the stream line. We visualize the direction by a special halo technique.

Traditionally, halo techniques have been used to increase depth perception, i.e. to enable the user to identify clearly which stream line is in front of other stream lines. Such a halo can be simply created by rendering a stream line twice as a line strip. The first pass renders the line as a line strip with a specific width and enabled depth buffer while the second pass renders the same line strip with a broader width and enabled depth buffer and a small z-offset. This way, the halo is drawn only at pixels that have not been covered by the first pass.

We change the second pass in order to visualize flow directions. Instead of rendering a durchgehend line strip, the second pass renders a *stippled line strip* which is fully supported by graphics hardware. A stippled line strip applies a bit pattern to the rendered line at pixel level. Every time when the bit pattern is 1, the pixel is drawn, and otherwise skipped. Bit patterns have typical

lengths of 16 bits which are repeated along the line.

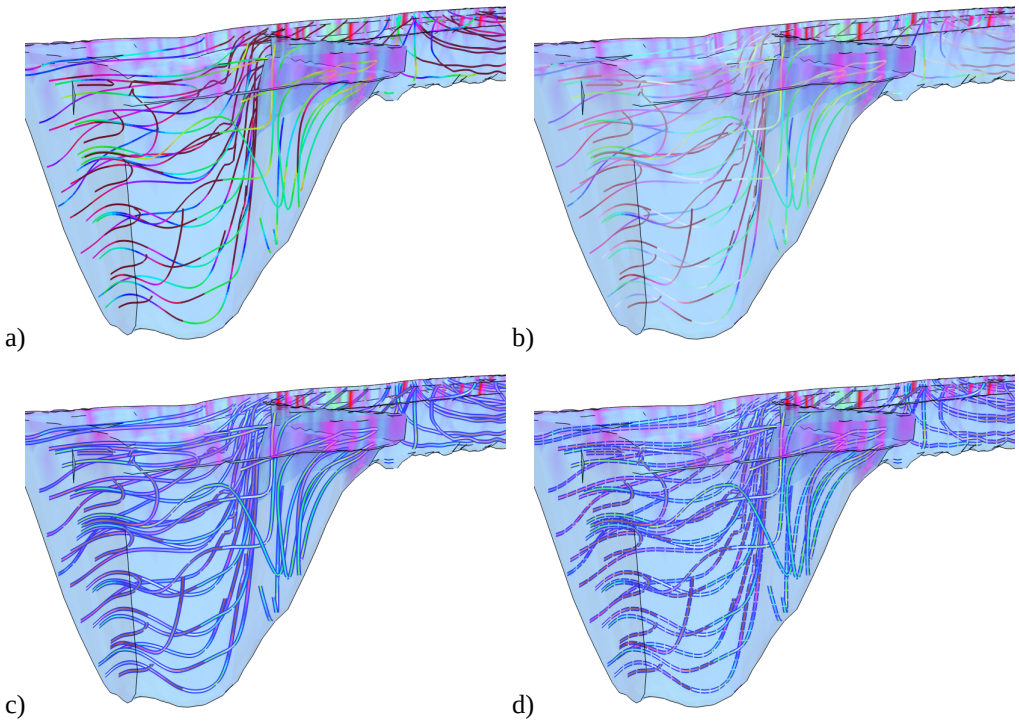


Figure 7.9: (a) A dense set of stream lines without lighting or halos. (b) With lighting. (c) With lighting and halos. (d) With lighting and stippled halos.

The stipple pattern for flow visualization is chosen to leave a one-bit (i.e. one-pixel) gap. The position of the gap moves one bit from frame to frame which results in the gap to move one pixel from frame to frame. The integration process stores the points of a stream line in the order of the flow direction. The strip is rendered in exactly this order. So, moving the stipple pattern perfectly coincides with the flow direction.

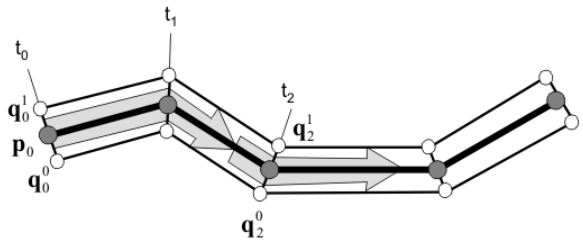


Figure 7.10: Textured arrows as stream line halo.

Stippled halos need animation for the visualization of flows directions. For still images, we replace the second pass by a geometric halo which is basically a quadriliteral strip with a texture

on it. The texture shows an arrow where pixels on the arrow are opaque while all other pixels of the texture are fully transparent.

Given a line segment $(\mathbf{p}_i, \mathbf{p}_{i+1})$, the quadriliteral for the halo is spanned by the sequence of point pairs $(\mathbf{q}_i^0, \mathbf{q}_i^1)$ for each point $\mathbf{p}_i$ of the stream line as shown in figure 7.10. The positions of the quadriliteral points can be simply calculated as the cross product between the normalized viewing direction $\mathbf{v}$ and the normalized vector $\mathbf{v}_i$ of the vector field at point $\mathbf{p}_i$:

$$\mathbf{q}_i^{0,1} = \mathbf{p}_i \pm g\ \mathbf{v} \times \mathbf{v}_i$$

where g is a scaling factor that influences the width of the halo. g can be increased with increasing depth, i.e. $g = g(\|\mathbf{o} - \mathbf{p}_i\|)$ for an observer position $\mathbf{o}$. A linear function often yields good visual results: $g(\|\mathbf{o} - \mathbf{p}_i\|) = s/e \times \|\mathbf{o} - \mathbf{p}_i\|$ with a user-controllable linearity factor s and the extent e of the dataset's bounding box.

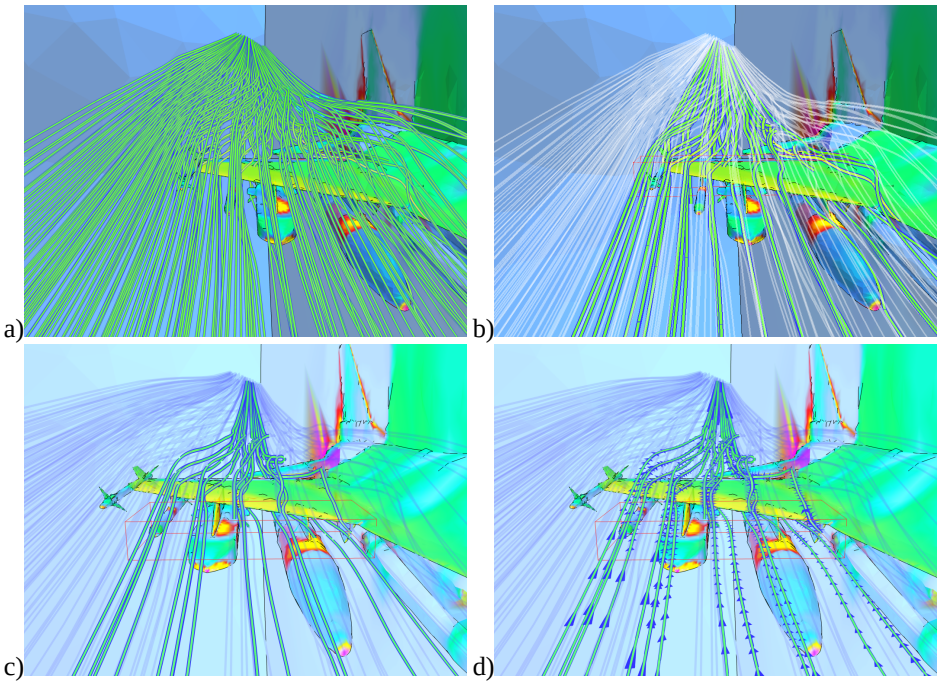


Figure 7.11: (a) The front tip of the wing of the F16 dataset with all streamlines visualized. (b) A small subset is specified by the red selection box and is rendered opaque. (c) A different view point and other coloring. (d) The halo is drawn which shows flow directions with arrow textures.

A texture coordinate t_i is assigned to each point pair $(\mathbf{q}_i^0, \mathbf{q}_i^1)$. The first pair $i = 0$ starts with the texture coordinate $t_0 = T$ which is now increased for subsequent pairs. For each pair $i > 0$, the texture coordinate is computed as $t_i = t_{i-1} + h\ \frac{1}{\|\mathbf{v_i}\|+1}$ with the not normalized vector $\mathbf{v_i}$ of the vector field at $\mathbf{p}_i$ and h as a scaling factor. If h depends on the depth $h = h(\|\mathbf{o} - \mathbf{p}_i\|)$, perspective foreshortening of the arrows can be avoided. Similar to g, a linear function yields good

visual quality. Because the texture coordinates t_i run out of the unit interval, the graphics API like OpenGL must repeat the texture for coordinates out of the unit interval.

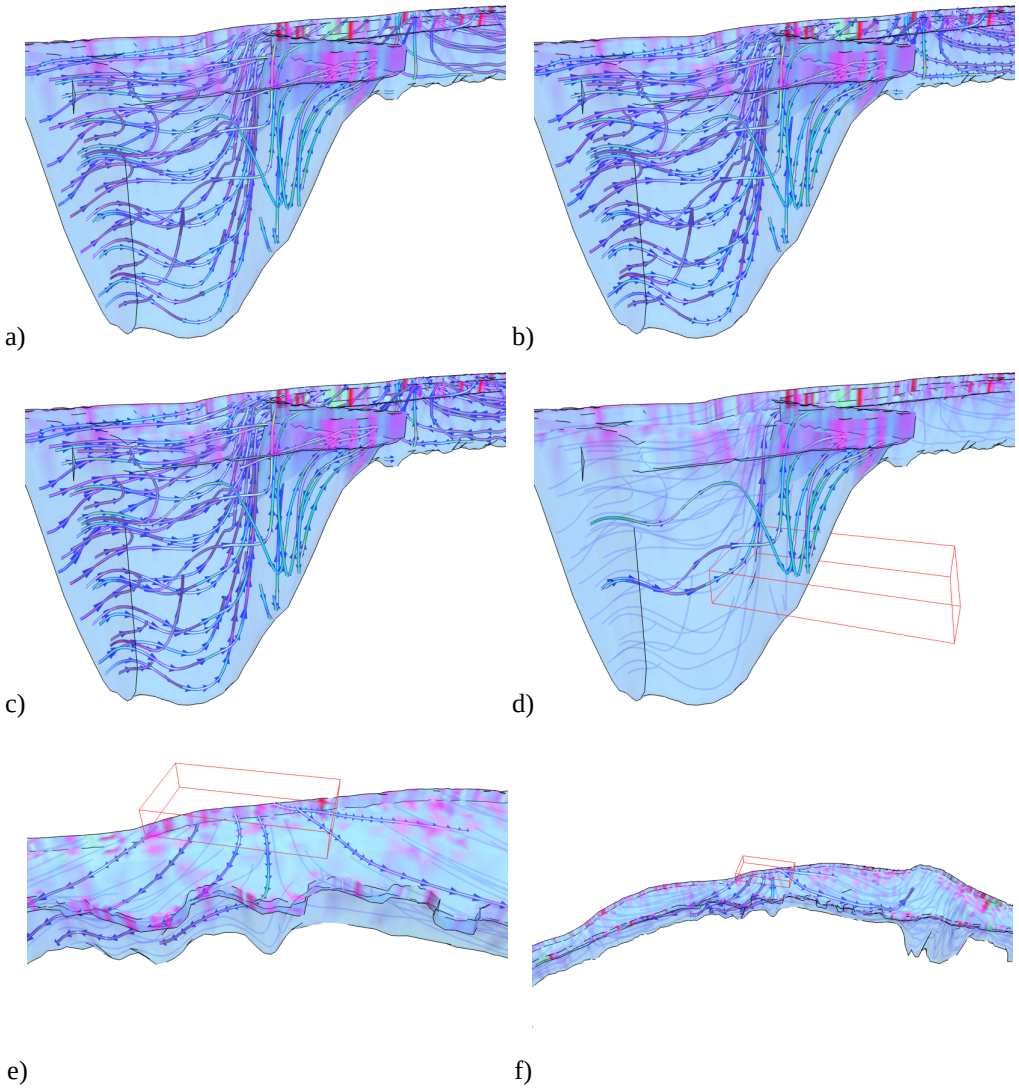


Figure 7.12: (a) Geometric halos without depth correction. (b) With depth correction. (c) With depth correction for textures also (notice the lengthened arrows). (d) Ghosting and halos. (e,f) The textures and halos scale with observer distance.

7.4 Diffusion Surfaces for Tensor Fields

A symmetric 3D tensor field is segmented into regions dominated by stream tubes and regions dominated by diffusion surfaces. The diffusion surfaces are integrated with a higher order Runge–Kutta scheme and approximated with a triangle mesh. Our main contribution is to steer the integration with a face-based coding scheme, that allows direct compression of the integrated diffusion surfaces and ensures that diffusion surfaces of any topology can be created.

Finally we sample the stream tubes and diffusion surfaces with points. The points from different entities are colored with different colors. We lit the points during rendering with a lighting model adapted to the tensor field. The resulting visualization of symmetric 3D tensor fields is sparse because of the sampling on points and allows for a deeper view inside the volumetric tensor field but also allows the simultaneous visualization of a dense set of tubes and surfaces.

7.4.1 Symmetric Tensors and Diffusion Surfaces

A lot of work has been done to visualize vector fields. Stream lines and stream surfaces are popular visualization techniques. A stream line is a curve where for every point on the curve the associated vector is tangent to the curve. One can imagine a stream line as the path that a particle takes through the vector field. Stream lines do not intersect each other except for points where the vector field vanishes or is undefined.

A stream surface is the path that a curve takes through the vector field and can be thought of as the dense collection of stream lines, all starting at a given curve.

The situation changes slightly if we look at symmetric 3D tensor fields. Throughout this paper we use the term tensor for symmetric 3D tensors. Symmetric 3D tensors play a great role in physics or medicine as for example diffusion tensors are symmetric 3D tensors. At every point a tensor field contains a (symmetric) tensor, i.e. a symmtric 3×3-matrix, instead of a single vector.

Eigenvector decomposition of the tensor is a popular approach to analyze a tensor. A symmetric tensor can always be decomposed into a diagonal matrix Λ with the three eigenvalues $\lambda_1 \geq \lambda_2 \geq \lambda_3$ on the diagonal and an orthonormal rotation matrix V with $VV^t = \mathbf{1}$, with the unit matrix $\mathbf{1}$:

$$\begin{aligned} \forall T \in \mathbf{R}^{3\times 3} \quad &\text{with} \quad T = T^t : \\ \exists V, \Lambda \in \mathbf{R}^{3\times 3} \quad &\text{with} \quad VV^t = \mathbf{1}, \Lambda_{ij} = 0 \Leftarrow i \neq j : \\ &\qquad\quad T = V\Lambda V^t. \end{aligned}$$

The columns $v_i \stackrel{def}{=} V_{.i}$ of V form an orthonormal basis of $\mathbf{R}^3$ and are called the eigenvectors. The combination of eigenvectors and eigenvalues (V, λ) is called the eigensystem of the tensor. If a unit sphere is scaled in the direction of the eigenvectors with the eigenvalues we obtain an ellipsoid that can be used to visualize the tensor. In the case of diffusion tensors the ellipsoids describe for any possible direction the rate of diffusion. Particles would have to be traced in all possible directions with speeds given by the ellipsoid. A diffusion tensor can be imagined as a

description of how a spherical water drop is diffused into an ellipsoid. The terms stream line and stream surface can therefore not so easily applied to tensor fields.

Diffusion tensors often degenerate over large regions, such that their ellipsoids look like cigars or like pancakes. In the case of a cigar we speak of linear anisotropy, when one eigenvalue dominates the other two. In the other case we speak of planar anisotropy and two eigenvalues are much larger than the last one. If all eigenvalues are of similar size the corresponding region of the tensor field is called isotropic.

The eigenvector with the largest eigenvalue is called the major eigenvector, the eigenvector with the smallest eigenvalue is called the minor eigenvector, and the eigenvector with the medium eigenvalue is the medium eigenvector.

Given the anisotropy of tensors, the domain of a tensor field can be partitioned into linear, planar, and isotropic regions. Every region only contains tensors of the one specific type of anisotropy.

Within linear regions, a tensor field can be interpreted as a vector field formed by the major eigenvectors. Thus, we can define *stream tubes* as tubes whose middle axis is a stream line. Every point on the stream line is tangential to the major eigenvector of the tensor at this point. The cross section of the tube is defined by the medium and minor eigenvectors which are perpendicular to the major eigenvector.

Within planar regions, a *diffusion surface* can be defined as surface whose tangential plane for every point is the plane defined by the major and medium eigenvectors, i.e. it is normal to the vector field of the minor eigenvectors. We use the term diffusion surface here because the term stream surface may be misleading to what a stream surface is for a vector field.

We segment from the symmetric 3D tensor field regions dominated by stream tubes and regions dominated by diffusion surfaces. We reconstruct a dense set of stream tubes and diffusion surfaces and point sample them in way that the distance between two points is inversely proportional to the diffusion rate. Thus the points are closely spaced along a stream tube and sparsely orthogonal to it. On a diffusion surface the points are closely spaced over the surface but the surfaces are further apart from each other. The human eye of the observer will automatically merge close points to tubes and surfaces. The points from different entities are distinguished by their color. We lit the points during rendering with a lighting model adapted to the tensor field. The resulting visualization of symmetric 3D tensor fields is sparse because of the sampling on points and allows for a deeper view inside the volumetric tensor field but allows on the other hand the simultaneous visualization of a dense set of entities.

There hasn't been much work done yet to extract diffusion surfaces from tensor fields. Zhang et al. [ZDL02] have presented a technique to extract stream tubes and diffusion surfaces from volumetric diffusion tensor MR images. Stream tubes are extracted in linear regions and diffusion surfaces in planar regions. So stream tubes represent structures with primarily linear diffusion while diffusion surfaces represent structures with primarily planar diffusion. Additional information is encoded in the color and cross section of the stream tubes. Zhang et al. call the diffusion surfaces stream surfaces. I prefer the term diffusion surface to avoid confusion with the term stream surface from vector fields.

My approach of extracting diffusion surfaces is similar to [ZDL02]. That's why this approach is discussed in more detail. Zhang et al. generate a dense set of stream tubes and diffusion surfaces and cull them later using a set of metrics.

Linear regions are interpreted as vector fields formed by the major eigenvectors of the tensors. Thus the trajectory of a stream tube is a stream line through this vector field. The MR images are interpolated using tricubic B-Splines to get tensors not only at the sample points of the MR images. Zhang et al. generate seed points for every sample point within a linear region and jitter them within the voxel. The stream tube starts at a seed point and follows the major eigenvector field both forward and backward. An second-order Runge-Kutta integrator is used to track the stream line.

Diffusion surfaces are extracted from planar regions. The major and medium eigenvectors of a tensor at a point in space define the tangential plane of the surface at this point. Again, the seed points are placed into the voxels by jittering the sample points.

Starting from a seed point $\mathbf{v}$, six initial search directions are distributed evenly around $\mathbf{v}$. Every search direction is tracked and thus follows the shape of the surface. A triangle is created for every pair of neighboring edges. This first step creates a triangle fan consisting of six triangles.

From every vertex $\mathbf{u}$ new search directions are created by projecting the triangles that are adjacent to $\mathbf{u}$ onto the tangential plane $P(\mathbf{u})$ and the initial directions in $P(\mathbf{u})$ that are not covered by triangles. This is repeated for every newly generated vertex.

The new search directions of a vertex $\mathbf{u}$ are traced through a vector field which is defined as the linear combination of the normalized major and medium eigenvectors which lies on a Plane P_1 that is both perpendicular to the tangential plane $P(\mathbf{u})$ at $\mathbf{u}$ and contains the search direction.

The extension of the diffusion surface stops if it gets out of the data boundary, hits a low planar region, enters a region of low signal–to–noise ratio, or incurs a high curvature term.

While rendering, color is mapped onto the surface to represent the planar anisotropy.

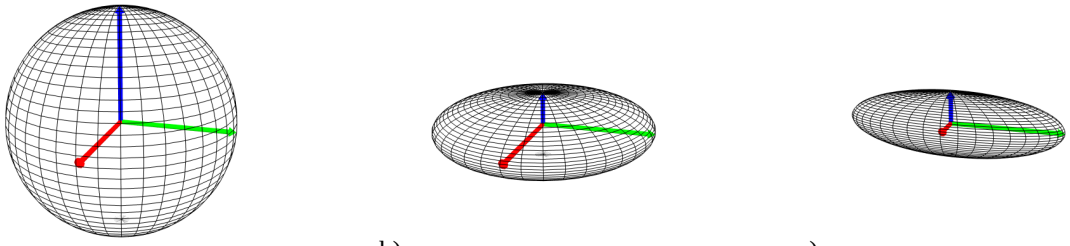
7.4.2 Point-Based Tensor Field Visualization

Volume Segmentation

The tensor field domain is partitioned into linear, planar, and isotropic regions. We use three quantities of a (diffusion) tensor to define this partition as suggested by [WPG+97]:

$$\begin{aligned} c_l &= \frac{\lambda_1 - \lambda_2}{\lambda_1 + \lambda_2 + \lambda_3} \\ c_p &= \frac{2(\lambda_2 - \lambda_3)}{\lambda_1 + \lambda_2 + \lambda_3} \\ c_s &= \frac{3\lambda_3}{\lambda_1 + \lambda_2 + \lambda_3} \end{aligned}$$

where c_l measures linear anisotropy, c_p planar anisotropy, and c_s isotropy. Note that $c_l + c_p + c_s = 1$. λ_i are the eigenvalues of the tensor with $\lambda_1 \geq \lambda_2 \geq \lambda_3$. The greater c_l is the more the ellipsoid



a) b) c)

Figure 7.13: A tensor can be classified as being isotropic ($c_l = 0$, $c_p = 0$, $c_s = 1$), planar ($c_l = 0$, $c_p = 0.61$, $c_s = 0.39$), or linear ($c_l = 0.57$, $c_p = 0$, $c_s = 0.43$).

looks like a cigar (and the smaller c_p and c_s are). Similarly, the greater c_p is the more the ellipsoid looks like a pancake. Finally, if c_s equals 1, the ellipsoid becomes a sphere (all eigenvalues are 1 and c_p and c_l are 0), see figure 7.13.

After this segmentation of the volume, we can trace stream lines in linear regions and diffusion surfaces in planar regions. We use thresholds on c_l and c_p to classify regions.

Distributing Points

We render stream tubes and diffusion surfaces as collections of points. Different colors are used to distinguish points from different entities. The tubes and surfaces shall be point sampled with the density described by the inverse of the diffusion rate given by the symmetric tensors.

Depending on the entity, that points were sampled from, the points are lit differently. Points from stream tubes are lit with the lighting model for lines as proposed by Zöckler et al. [ZSH96], whereas points on the diffusion surfaces are lit with the standard Blinn-Phong lighting model provided by OpenGL.

Stream tubes

Similar to [ZDL02] we look at the tensor field in linear regions as a vector field consisting of the major eigenvectors of the tensors. This vector field is traced.

We subdivide the volume into a regular grid which may be given automatically by the resolution of the image data. Otherwise, the resolution of the grid is specified explicitly.

To create stream tubes, we first create a stream line for every stream tube. This stream line is the trajectory of the latter stream tube.

We place a seed point into every grid cell. The tensor of this seed point needs to have linear anisotropy. We integrate a stream line starting at the seed point into both forward and backward direction using a second-order Runge-Kutta integrator. A stream line consists of a list of ordered points p_i and is linearly approximated as a line segment between two successive points p_{i-1} and p_i.

We want to place the points along a stream line such that the density of the points on a stream line represents the linear anisotropy of the tensors involved as described above.

To realize this behavior, we control both the arc length of the integration process and the step width of the second-order Runge-Kutta integrator.

The integration of the stream line stops if any of these cases happens:

- Outside of data volume.
- Left region of linear anisotropy $c_l > C_l$ where C_l is the threshold for linear anisotropy.
- Extended point is "too" close to a previously calculated point.

The third point is motivated by artificial datasets where a stream line may be a (closed) circle. In order to stop the integrating process, the integrator needs to check if it reaches a part of the stream line that was previously integrated. For a fast local access to the points and line segments of the stream line, we sort the extended points into an octree. The integrator just needs to look up the octree to find proximate points.

A stream tube follows the trajectory defined by the stream line. Our approach is to render the (extended) points of the stream line only instead of rendering the whole tube around the stream line.

Diffusion surfaces

A crucial point for our visualization technique is to distribute the points across diffusion surfaces. We want the points to be distributed according to the diffusion rate over the diffusion surface. The higher the diffusion rate, the closer have to be the points. Remember that we want to render points instead of shaded surfaces. The human eye connects points that lie closer together and therefore follows automatically the more likely diffusion direction.

Although we are only interested in rendering points, for the integration of the diffusion surface it is advantageous to also build a triangle mesh to ensure a proper diffusion surface. Furthermore the connectivity information allows for smoothing and successive remeshing steps.

From a mathematical point of view, a diffusion surface is defined as a normal surface. Given a continuous vector field V and a point $\mathbf{p} \in \mathbf{R}^3$. A surface which contains $\mathbf{p}$ and whose field of normals is parallel to V is called a *normal surface* at $\mathbf{p}$ for V.

The Forbenius' theorem states if a normal surface is well-defined:

A normal surface of a differentiable vector field V are defined iff the vector field V is orthogonal to its rotation $rot(V)$, i.e.

$$V \perp rot(V).$$

The minor eigenvalues in planar regions form a vector field N of normals and the normal surfaces of N are called *diffusion surfaces*. In order to define a mathematical solid integration process for diffusion surfaces, the scalar product between the normal field N and its rotation $rot(N)$ is controlled by a threshold δ. If the scalar product exceeds δ, the integration process of § 7.4.3 is stopped.

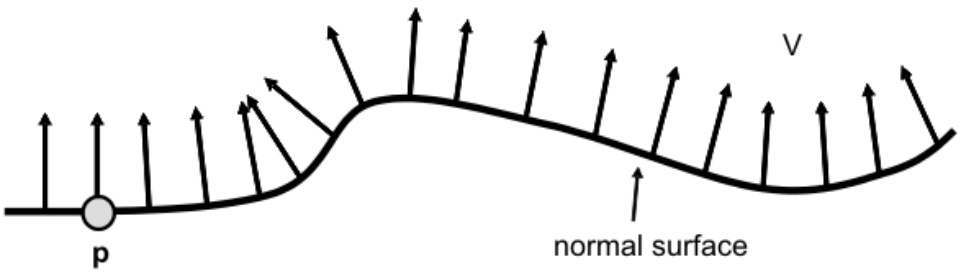


Figure 7.14: The normal surface at $\mathbf{p}$ *for the vector field* V.

7.4.3 Diffusion Surface Integration

As the diffusion surface has to be manifold with border, the integration process that approximates the diffusion surface via a triangle mesh is very similar to the encoding or rather decoding process of a faced-based compression scheme. We used exactly the same building scheme to build the triangular diffusion surface. There are three advantages with this approach. Firstly, we can re-use the minimum set of building operations that allow to create manifold meshes of arbitrary genus. Secondly, we can re-use the data structures used for face-based coding such that the implementation of the diffusion surface integrator becomes very simple. And the third advantage is that we can directly encode the triangular diffusion surface into a space efficient representation, such that we can easily build in-core a large number of diffusion surfaces of high resolution.

We used a face-based compression scheme similar to the cut-border machine [GS98] and the edge-breaker [Ros98] for coding and to steer the integration. A short review of the method and the basic building operations are given in the next subsection.

Face-Based Coding of Triangle Meshes

Face-based coding techniques compress triangle meshes which consist of a list of vertices and a list of triangles, each triangle containing three vertex indices and the indices of the edge-adjacent triangles.

The schemes are based on a region growing traversal of the triangle mesh. The traversal begins for example with an arbitrary seed triangle. The border of the growing region is called the *cut-border*. It divides the mesh into the *inner* and the *outer part*, which contain the already processed and the untouched triangles respectively. Triangles are added to the inner part at a distinguished current cut-border edge which is called the *gate*. After each addition of a triangle the gate moves on to another cut-border edge, until all triangles of an edge-connected component of the triangle mesh have been compressed. This is done for each edge-connected component. The choice of the next gate location defines the *traversal order* and steers how the cut-border grows over the mesh.

The face-based coding scheme encodes a bit-code each time a new operation is added. The decoding performs the same traversal and builds the face according to the encoded operation. The different possible operations by which the next triangle is incorporated into the inner part at the

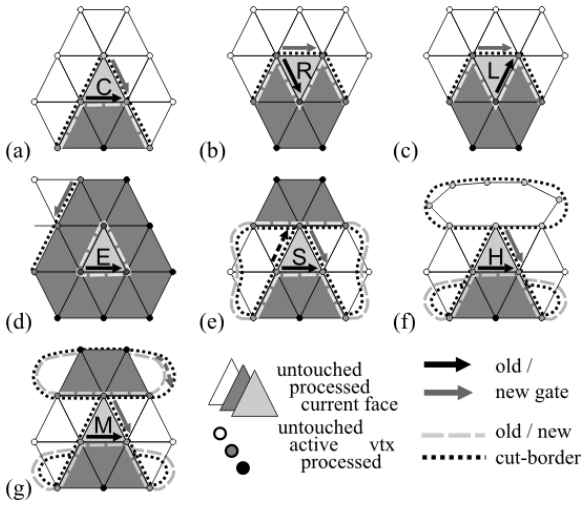


Figure 7.15: The different cut-border operations in which the processed triangle can be incident to the cut-border.

gate are illustrated in figure 7.15. The *center* operation C in (a) adds a new triangle to the growing region that is incident only to the old gate and to a new vertex. The gate is moved to the right newly added cut-border edge such that it cycles around its target vertex. The current face in the *right/left* operation R/L in (b/c) is incident to the gate *and* the next/previous edge on the cut-border. The neighborhood of the pivot vertex is closed and a new pivot vertex is chosen with the new gate location. In the *end* operation E in (d) all edges of the current face are incident to the cut-border and the cut-border closes. The other growing operations describe cases when the third vertex of the current face is on the cut-border. (e) shows the *split* operation S, where the cut-border grows into itself and is split into two loops with two gate locations. One cut-border loop is pushed onto a stack and processed after the other one is eliminated by an *end* operation. In order that the decoder can replay the split operation the position of the third vertex relative to the gate is encoded. The *hole* operation H (f) merges the current cut-border with a border loop. We will handle border loops in a different way as done by the cut-border machine [GS98]. We encode a border operation B, whenever the gate is a border edge of the mesh. As triangle meshes describe two dimensional surfaces in three dimensional space, two cut-border loops can grow together again, actually once for each handle of the triangle mesh. The operation which unifies two loops is called *merge* operation M (g). It merges the current cut-border loop with another cut-border loop and takes two indices, the index of the other cut-border loop and the location inside that other loop.

The cut-border data structure consists of a stack of doubly linked lists of cut-border edges. Each cut-border edge stores the indices of its start vertex and of its adjacent triangle in the inner part. The initialization creates a cut-border with three edges. Each C,S or M operation inserts a cut-border edge after the gate. The R and L remove the next or previous cut-border edge and the E operation closes the loop on top of the stack.

Cut-Border Based Surface Construction

As the face-based coding scheme can encode any manifold mesh, one can also build any triangulated diffusion surface during the integration.

The input to our diffusion surface integrator is a seed location in a planar region of the volumetric domain of the tensor field. We integrate the diffusion surfaces with the face-based building operations C,L,R,E,S,M,B. Opposed to the face-based scheme we start the building process with a single edge and initialize the cut-border to a loop of two cut-border edges around the edge like the face-fixer proposed by Isenburg [IS00]. This first edge is created by integrating from the seed location in the direction of the major eigenvector of the tensor field. The integration of a new edge is described in the next subsection in detail.

The diffusion surface is built from the initial cut-border by extending it at one of the cut-border edges, which is called the gate. The choice of the gate steers how the cut-border grows the diffusion surface and is described in subsection 7.4.3 in detail.

Input: seed location $\mathbf{x}$

integrate edge from $\mathbf{x}$ to $\mathbf{y}$
init cut-border to $(\mathbf{x}, \mathbf{y})$
while cut-border not empty
    choose gate
    decide operation
    apply operation to cut-border

Figure 7.16: Structure of the integration algorithm.

We summarized the integration algorithm in figure 7.16. After the cut-border has been initialized from the seed location we successively select a gate edge, decide which of the operations C,L,R,E,S,M,B to perform and apply it to the cut-border until no cut-border edges are remaining. This can either happen, when the diffusion surfaces closes up or when all cut-border edges were transformed into border edges of the mesh by a B operation at the boundary of the planar domain of the seed location. Figure 7.17 illustrates the growing process for a spherical diffusion surface.

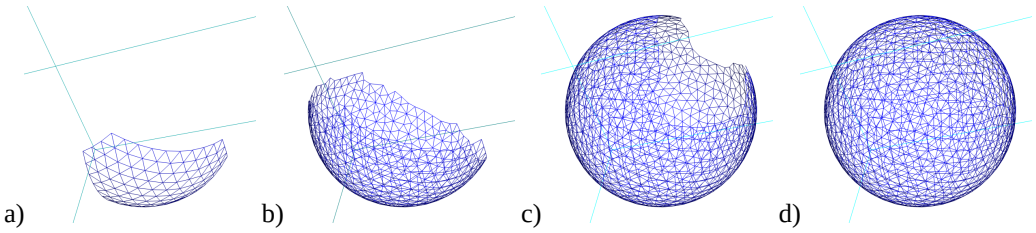


Figure 7.17: Illustration of how a diffusion surface is grown.

In this subsection it remains to explain how we decide for a given gate location, which operation

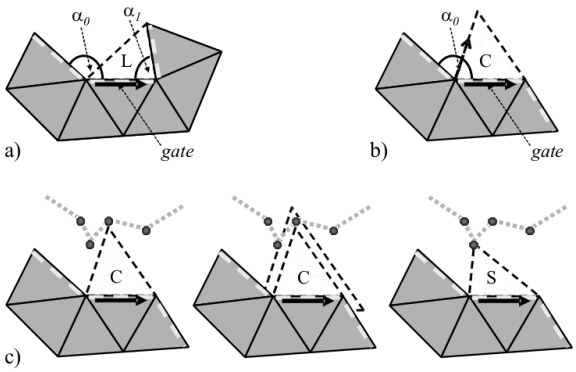


Figure 7.18: a) angles measured to decide for L, R or E operations b) integration direction c) check for split or union

has to be performed. We first propose one of the operations E,L,R,B or C and then check for E,L,R or C if we should not have performed an S or M instead.

First we decide if we should propose an L,R or E operation based on the exterior angles between the gate edge and the previous or next edge on the cut-border as depicted in Figure 7.18 a). If one of the angles α_0 or α_1 is smaller than a threshold angle, that we set to seventy degrees, we decide for an L or R operation. When the length of the current cut-border loop is only three, we decide for an E operation instead of L or R.

If both angles are larger than the threshold, we try to propose a B operation by trying to integrate the left edge of the new triangle as shown in Figure 7.18 b). The direction for integration should be in the planar case always sixty degrees. For curved surfaces this is not possible anymore. Here we chose to subdivide the angle α_0 into k equal parts such that the resulting angle is closest to sixty degrees. In the example of Figure 7.18 b) k is two such that we chose the direction $\alpha_0/2$ away from the gate.

The integration process is in the next subsection. It returns a target location or reports failure if the integration left the planar domain. In case of failure we decide for a border operation B, that will always be performed. Otherwise we can construct with the target point a new triangle labeled with C in the Figure 7.18 b).

Now we either perform a border operation or propose a new triangle added by E,L,R or C. This triangle can intersect a distant cut-border part as illustrated in Figure 7.18 c). Here we should rather perform a S or M operation with the vertex closest to the gate. Whether S or M is performed can be decided by the cut-border data structure from the vertex to which the gate will be connected. To find out if an S or M has to be performed, we entered all cut-border edges in an octree and checked for the proposed E,L,R and C triangles if they cover other cut-border edges. In order to also connect to close outside cut-border vertices we enlarge the proposed triangles by 30% before we checked in the octree. From the covered cut-border edges and vertices we selected the cut-border vertex closest to the center of the gate and proposed an S or M operation. See Figure 7.18 c) for

an illustration of this process.

Integration of New Edges

For every search direction, we define the search plane as the plane that is orthogonal to the tangential plane at the start point, contains the search direction vector and the start point itself. This approach is similar to [ZDL02].

Every search plane defines a vector field along which we integrate from the start point. A vector of this vector field is, simply spoken, defined by the cut of the search plane and the plane which is spanned by the major and medium eigenvector of a tensor. The length of this vector is just the distance between the location of the tensor and the cut point of the search plane with the ellipse defined by the major and medium eigenvector, see figure 7.19.

To perform this operation quickly, we use the following simple mathematics. Let n be the normal of the current search plane, and T be the tensor matrix at the current point. The vector field is then defined by the following operations:

$$\begin{aligned} \tilde{n} &= T^t n \\ n_\perp &= \begin{pmatrix} -\tilde{n}_1 \\ \tilde{n}_0 \\ 0 \end{pmatrix} \\ v_t &= T n_\perp \end{aligned}$$

We transform the normal vector of the search plane into the coordinate system defined by the eigenvectors. That is just a matrix–vector multiplication. Because we want the vector to lie in the plane of the two largest eigenvectors, we project the vector into this plane by setting the last coordinate to 0. Then, an orthogonal vector is set up by multiplying the vector to a 90 rotational matrix. This can be done explicitly by just changing the coordinates. This orthogonal vector is then transformed back into the world coordinate system to be the vector of the vector field within the search plane.

The step width for the integration process of one search direction is adapted to the eigenvalues in order to distribute the points according to the anisotropy of the tensor field

$$h = C \sqrt{\sum_i w_i^2 n_{\perp i}^2}$$

where $n_\perp$ is the vector orthogonal to the normal vector of the search plane as defined above, h is the step width, and C is a user-controlled constant which can further influence the density of the points. The weights w_i are chosen to represent the proportional behavior to the anisotropy as

$$w_i = \frac{1}{\lambda_i}$$

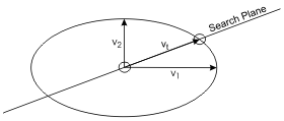


Figure 7.19: A vector of the vector field defined by the cut of the search plane and the ellipsoid within the plane of the two largest eigenvectors.

Traversal Order

In order to avoid a large number of S or M operations we used a similar strategy as proposed by Alliez and Desbrun [AD01b]. The method is based on the observation that S operations arise very seldom at cut-border edges that are in a convex region. Therefore we measured for each cut-border edge the two exterior angles α_0 and α_1. The smaller these angles are, the more convex is the region around this cut-border edge. As it is already fine if one of the angles is small, we sorted the cut-border edges according to the minimum if the exterior edges into a priority queue. Each time a new gate has to be chosen, we extracted the most convex from the priority queue. After each basic operation we updated also the cut-border edges adjacent to newly added or removed ones and re-positioned them in the queue. Figure 7.17 illustrates that the cut-border stays nicely shaped during the integration process.

7.4.4 Results & Analysis

We tested our algorithm with tensor fields that include singularities which need to be bypassed by the integration and meshing process. We created artificial tensor fields to simulate different behavior and used MRI datasets for stream line tracking.

Artificial Tensor Fields

The Spiral example has singularities on the z axis. The tensors have planar anisotropy around the z axis, see figure 7.20. The diffusion surfaces are rendered as triangular meshes to show how the mesh generation follows exactly the diffusion surface and thereby bypasses the singularities.

A dataset similar to sphere, see figure 7.21, demonstrates the usefulness and power of the point–rendering approach. The interior stream lines are completely enclosed by a mesh but they remain visible.

The Sinosoidal example shows both stream tubes and stream surfaces within a dataset that varies anisotropy very frequently. Figure 7.21 shows a simple preview of the dataset displaying the major, medium, and minor axes of the tensors within a 10x10x10 grid. The color encode the different anisotropies. Green is planar anisotropy, red linear, blue isotropy, and gray is undefined. Figure 7.21 shows the point–rendered stream tubes and diffusion surfaces.

The crucial part of the performance of our algorithm is the integration process itself. Thus the step width of the integration step is the limiting factor and needs to be adapted very carefully to

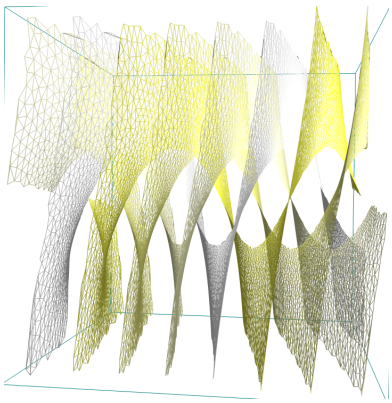


Figure 7.20: The diffusion surface is a spiral around the singularities at the z axis.

the anisotropy of the tensor field.

Note that the distribution of the points over the diffusion surface may not be optimal. We did not implement an optimization algorithm for placing the points. We can ensure that the whole diffusion surface is extracted because our approach starts with an initial triangle and grows it's border until it reaches a non–linear region, and that the distance between points along edges that were integrated is optimal in terms of that this distance accords to the diffusion rate along this edge. But there may be edges that were not integrated but artificially inserted by the cut–border operations. The optimal distribution of points over a diffusion surface is topic of further research.

DT-MRI Tensor Fields

Diffusion tensors play an important role in medicine where they can be calculated with magnet resonance imaging (MRI) technology to produce diffusion tensor MRI (DT-MRI).

The myeline structures which surround nerve fibers are impervious to water and cause fluids to diffuse into the directions of nerve fibers. In reverse, the diffusion directions of water reveals information about the tracks of nerve fibers.

We compute water diffusion tensors of the human brain out of seven MRI scans by using the TEEM library which is public available. The resulting volumes are regular grids with a typical resolution of $192 \times 192 \times 65$. At each grid point, a diffusion tensor is stored. Diffusion tensors are symmetric and semi-definite tensors of rank 2. Hence, the algorithm presented in this section can be applied. In regions of linear anisotropy, water diffuses mainly into one direction, i.e. along a nerve fiber.

A dense set of stream lines is computed starting at uniformly placed seed points in regions of linear anisotropy. A stream line is integrated along the vector field formed by all largest eigenvectors. In order to reduce noise inherent present in MRI datasets, a moving least squares filter smoothes all tensors. The computational expensive 3D integrals of the moving least squares filter are solved by randomized integration.

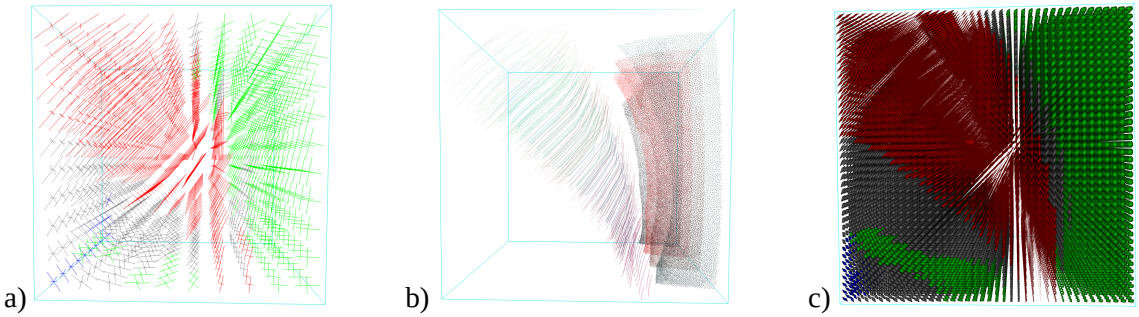


Figure 7.21: (a) A simple preview for datasets with symmetric tensors. Green is planar anisotropy, red linear. The axes of eigenvectors are shown. (b) Stream tubes and diffusion surfaces rendered as point clouds. (c) A classic approach for visualizing tensor fields is to render ellipsoids. Note the lack of visibility due to occlusion of the ellipsoids.

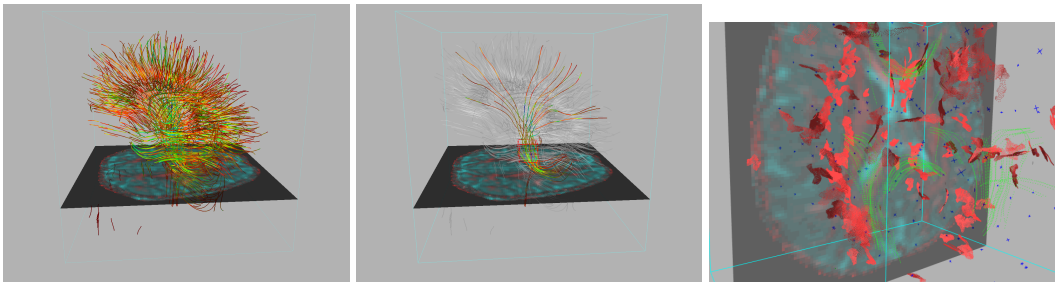


Figure 7.22: A dense set of stream lines has been extracted and is visualized together with a referencing slice through the dataset. Stream lines that pass regions of planar anisotropy can be extended by integrating diffusion surfaces in order to yield such dense sets.

The integration process of the stream lines terminates when a region of non-linearity is reached, i.e. a region where no eigenvalue dominates the other two eigenvalues. If a stream line terminates because a planar region has been detected (which can happen at the crossings of stream line bundles), we start to integrate a diffusion surface. At the boundary of the diffusion surface, we again start to integrate stream lines.

Chapter 8

Conclusions and Future Work

Die Wissenschaft fängt eigentlich erst da an, interessant zu werden, wo sie aufhört.
- Justus von Liebig

The presented rapid simplifier is based on a separated simplification of the triangular boundary and a point sampling of the interiour together with a final tetrahedralization. The tetrahedralization step can be avoided if a table is built which maps each vertex index of the original mesh to its new vertex index in the simplified mesh. The boundary simplification adds all vertices that it changes to the table while the interiour sampling adds all vertices that are changed by the sampling. Now, a final loop over all tetrahedra can map all four original indices of a tetrahedron onto their new indices. If a tetrahedron is assigned four different new vertex indices, the tetrahedron remains valid and is part of the approximating mesh. Otherwise, the tetrahedron can be discarded. Such an approach may extend [LT98] to tetrahedral meshes and needs to add some logic to avoid tetrahedral flips. Finally, the loop over all tetrahedra can compute quadrics and spread them over the octree cells. This way, each octree cell stores a single quadric and the sampling point of each cell can be placed quadric-optimal.

Only a few algorithms exist to represent a tetrahedral mesh as a compressed progressive mesh. In contrast to a full–blown multi–resolution model, the mesh can be refined uniformly only but enables better compression rates [PRS99]. While the tetrahedral cut-border as the best single resolution compression scheme achieves about 2 bits per tetrahedron for connectivity compression, the progressive representations achieve about 4 bits per tetrahedron for manifold meshes. Recently, Isenburg et al. applied their streaming meshes approach to tetrahedral meshes but achieved compression of about 3 bits per tetrahedron for connectivity. The out-of-core data structure of § 6.1 might achieve better compression rates for huge meshes because the simple compressor of § 6.3 achieves already 4 bits per tetrahedron without any arithmetic (or other) coding. Finally, the compression of geometry is still worse for tetrahedral meshes leaving a big room for improvements. This is mainly caused by the prediction techniques which work well for polygonal datasets but fail to predict well for tetrahedral meshes.

Many simulations run on hybrid meshes which often need to be transformed into pure tetrahe-

dral meshes for interactive visualizations. As example, many hexahedral meshes are converted to tetrahedral meshes which produces for each hexahedron about 6 tetrahedra. Nevertheless, many hexahedral meshes contain small parts of unstructured tetrahedral elements anyway. The automatic construction of multi–resolution representations for such meshes is a field of research that has not been covered in depth yet during the last decades.

Last but not least, the visualization of volume meshes needs to explore techniques which allow for a high–quality visualization with simple algorithms for non–manifold and potentially self–intersecting meshes. Here, point based approaches like § 7.2 seem to be advantegeous as they are easy to create and handle.

Bibliography

[ACSD05] P. Alliez, M. Cohen-Steiner, and M. Desbrun. Variational tetrahedral meshing. In *ACM SIGGRAPH 05*, pages 617–625, 2005.

[AD01a] Pierre Alliez and Mathieu Desbrun. Progressive encoding for lossless transmission of 3d meshes. In *ACM SIGGRAPH 01*, pages 195–202, 2001.

[AD01b] Pierre Alliez and Mathieu Desbrun. Valence-driven connectivity encoding for 3d meshes. In *Eurographics 01 Conference Proceedings*, pages 480–489, 2001.

[Ake93] Kurt Akeley. Realitygraphics engine. In *Computer Graphics, Proceedings of ACM SIGGRAPH*, pages 109–116, 1993.

[AS98] Pankaj. K. Agarwal and Subhash Suri. Surface approximation and geometric partitions. *SIAM Journal of Computing*, 27:24–33, 1998.

[ASCE02] N. Aspert, D. Santa-Cruz, and T. Ebrahimi. Mesh: Measuring error between surfaces using the hausdorff distance. In *Proceedings of the IEEE International Conference on Multimedia and Expo (ICME 02)*, pages 705–708, 2002.

[Bie06] Jacobo Bielak. The quake project. http://www.cs.cmu.edu/q̃uake, 2006.

[CBPS06] S. P. Callahan, L. Bavoil, V. Pascucci, and C. T. Silva. Progressive volume rendering of large unstructured grids. *IEEE Transactions on Visualization and Computer Graphics (Proceedings Visualization / Information Visualization 2006)*, 12(5):1307–1314, Sept/Oct 2006.

[CCM+00] P. Cignoni, D. Constanza, C. Montani, C. Rocchini, and R. Scopigno. Simplification of tetrahedral meshes with accurate error evaluation. In *Proceedings of the conference on IEEE Visualization '00*, pages 85–92, 2000.

[CDFL+04] P. Cignoni, L. De Floriani, P. Lindstrom, V. Pascucci, J. Rossignac, and C. Silva. Multi–resolution modeling, visualization and streaming of volume meshes. *Eurographics '04 Tutorial Notes*, 2004.

[CDFM+04] P. Cignoni, L. De Floriani, P. Magillo, E. Puppo, and R. Scopigno. Selective refinement queries for volume visualization of unstructured tetrahedral meshes. *IEEE Transactions on Visualization and Computer Graphics*, 10:29–45, Feb 2004.

[CGG+03] P. Cignoni, F. Ganovelli, E. Gobetti, F. Marton, F. Ponchio, and R. Scopigno. Bdam: Batched dynamic adaptive meshes for high performance terrain visualization. *Computer Graphics Forum*, 22(3):505–514, September 2003.

[CGG+04] Paolo Cignoni, Fabio Ganovelli, Enrico Gobbetti, Fabio Marton, Federico Ponchio, and Roberto Scopigno. Adaptive tetrapuzzles: efficient out-of-core construction and visualization of gigantic multiresolution polygonal models. *ACM Transactions on Graphics*, 23(3):796–803, 2004.

[CGG+05] P. Cignoni, F. Ganovelli, E. Gobbetti, F. Marton, F. Ponchio, and R. Scopigno. Batched multi triangulation. In *IEEE Visualization 05*, pages 207–214, 2005.

[CICS05] S. P. Callahan, M. Ikits, J. L. D. Comba, and C. T. Silva. Hardware-assisted visibility sorting for unstructured volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 11(3):285–295, 2005.

[CKM+99] J. Comba, J. T. Klosowski, N. Max, J. S. B. Mitchell, c. T. Silva, and P. Williams. Fast polyhedral cell sorting for interactive rendering of unstructured grids. *Computer Graphics Forum (Proceedings of Eurographics 99)*, 18(3):369–376, 1999.

[CL03] Y. Chiang and X. Lu. Progressive simplification of tetrahedral meshes preserving all isosurface topologies. In *Computer Graphics Forum (Special Issue for Eurographics '03)*, volume 22, pages 493–504, 2003.

[Cla76] James E. Clark. Hierarchical geometric models for visible surface algorithms. *Communications of the ACM*, 19(2):547–554, October 1976.

[CM02] Prashant Chopra and Jörg Meyer. Tetfusion: an algorithm for rapid tetrahedral mesh simplification. In *Proceedings of the conference on Visualization '02*, pages 133–140, 2002.

[CMO97] Jonathan Cohen, Dinesh Manocha, and Marc Olano. Simplifying polygonal models using successive mappings. In *Proceedings of IEEE Visualization 97*, pages 395–402, 1997.

[CMRS03] P. Cignoni, C. Montani, C. Rocchini, and R. Scopigno. External memory management and simplification of huge meshes. *IEEE Transactions on Visualization and Computer Graphics*, 9:525–537, Nov 2003.

[COM98] Jonathan Cohen, Marc Olano, and Dinesh Manocha. Appearance-preserving simplification. In *Proceedings of SIGGRAPH 98*, pages 115–122, 1998.

[CRS98] P. Cignoni, C. Rocchini, and R. Scopigno. Metro: Measuring error on simplified surfaces. *Computer Graphics Forum*, 17(2):167–174, 1998.

[CVM+96] Jonathan Cohen, Amitabh Varshney, Dinesh Manocha, Greg Turk, Hans Weber, Pankaj Jr. Agarwal, Frederick P. Brooks, and William Wright. Simplification envelopes. In *Proceedings of SIGGRAPH 96, Computer Graphics Proceedings, Annual Conference Series*, page 119128, 1996.

[DDF02] E. Danovaro and L. De Floriani. Half–edge multi tessellation: a compact representation for multiresolution tetrahedral meshes. In *Proceedings 1st International Symposium on 3D Data Processing Visualization and Transmission*, pages 494–499. IEEE Computer Society, 2002.

[DDFM+05] E. Danovaro, L. De Floriani, P. Magillo, E. Puppo, D. Sobrero, and N. Sokolovsky. The half-edge tree: A compact data structure for level-of-detail tetrahedral meshes. In *Proceeding of the International Conference on Shape Modeling*, pages 15–17, 2005.

[DDFMP02] E. Danovaro, L. De Floriani, P. Magillo, and E. Puppo. Data structures for 3d multi–tessellations: an overview. In *Proceedings of the Dagstuhl Scientific Visualization Seminar*. Kluwer Academic Publishers, May 2002.

[DEGN99] T. K. Dey, H. Edelsbrunner, S. Guha, and D. V. Nekhayev. Topology preserving edge contraction. *Publ. Inst. Math. (Beograd) (N.S.)*, 66:23–45, 1999.

[DFKP04] L. De Floriani, L. Kobbelt, and E. Puppo. A survey on data structures for level-of-detail models. *Advances in Multiresolution for Geometric Modelling*, pages 49–74, 2004.

[DWS+97] M. Duchaineau, M. Wolinsky, D. Sigeti, M. Miller, C. Aldrich, and M. Mineev-Weinstein. Roaming terrain: Real-time optimally adapting meshes. In *IEEE Visualization 97*, pages 81–88, 1997.

[Ede01] Herbert Edelsbrunner. *Geometry and Topology for Mesh Generation*. Cambridge University Press, 2001.

[ESV99] J. El-Sana and Amitabh Varshney. Generalized view–dependent simplification. *Computer Graphics Forum*, 18(3):83–94, 1999.

[FL79] Robert Fowler and James J. Little. Automatic extraction of irregular network digital terrain models. In *Computer Graphics (Proceedings of SIGGRAPH 79)*, volume 13, pages 199–207, 1979.

[Gar99] Michael Garland. Quadric-based polygonal surface simplification. Technical report, Ph.D. thesis, Carnegie Mellon University, CS Dept. Tech. Rept. CMU-CS-99-105, 1999.

[GBK03] S. Gumhold, P. Borodin, and R. Klein. Intersection free simplification. In *The 4th Israel-Korea Bi-National Conference on Geometric Modeling and Computer Graphics*, pages 11–16, 2003.

[GDL+02] B. Gregorski, M. Duchaineau, P. Lindstrom, V. Pascucci, and K. Joy. Interactive view-dependent rendering of large isosurfaces. In *Proceedings of IEEE Visualization 2002*. IEEE Computer Society, Oct 2002.

[GGS99] Stefan Gumhold, Stefan Guthe, and Wolfgang Straßer. Tetrahedral mesh compression with the cut-border machine. In *Proceedings of the Visualization '99 Conference*, pages 51–58, 1999.

[GH97] M. Garland and P. Heckbert. Surface simplification using quadric error metrics. In *ACM Transactions on Graphics (SIGGRAPH)*, pages 209–216, 1997.

[GMT98] John. R. Gilbert, Gary L. Miller, and Shang-Hua Teng. Geometric mesh partitioning: Implementation and experiments. *SIAM Journal on Scientific Computing*, 19(6):2091–2110, 1998.

[GRS+02] Stefan Guthe, Stefan Roettger, Andreas Schieber, Wolfgang Straßer, and Thomas Ertl. High-quality unstructured volume rendering on the pc platform. In *Graphics Hardware Workshop 2002*, pages 1–8, 2002.

[GS98] Stefan Gumhold and Wolfgang Straßer. Real time compression of triangle mesh connectivity. In *SIGGRAPH 98 Conference Proceedings*, pages 133–140, 1998.

[Gui95] Andre Guiziec. Surface simplification with variable tolerance. In *Second Annual International Symposium on Medical Robotics and Computer Assisted Surgery*, pages 132–139, Nov 1995.

[GWGS02] Stefan Guthe, Michael Wand, Julius Gonser, and Wolfgang Straßer. Interactive rendering of large volume data sets. In *IEEE Visualization 2002*, pages 230–237, 2002.

[GZ05] Michael Garland and Yuan Zhou. Quadric-based simplification in any dimension. *ACM Transactions on Graphics*, 24(2), 2005.

[Ham94] B. Hamann. A data reduction scheme for triangulated surfaces. *Computer Aided Geometric Design*, 11(2):197–214, 1994.

[HDD+93] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle. Mesh optimization. In *Proceedings of ACM SIGGRAPH 1993*, pages 19–26, 1993.

[HG99] P. S. Heckbert and M. Garland. Optimal triangulation and quadric-based surface simplification. In *Computational Geometry: Theory and Application 14*, number 1-3, pages 49–65, November 1999.

[Hop96] Hugues Hoppe. Progressive meshes. *Computer Graphics*, 30(Annual Conference Series):99–108, 1996.

[Hop97] Hugues Hoppe. View-dependent refinement of progressive meshes. *Computer Graphics*, 31(Annual Conference Series):189–198, 1997.

[IG03] Martin Isenburg and Stefan Gumhold. Out-of-core compression for gigantic polygon meshes. In *ACM Transactions on Graphics*, volume 22, pages 935–942, 2003.

[IL05] Martin Isenburg and Peter Lindstrom. Streaming meshes. In *Proceedings of IEEE Visualization 2005*, pages 231–238, 2005.

[ILSS06] Martin Isenburg, Yuanxin Liu, Jonathan Shewchuck, and Jack Snoeyink. Streaming computation of delaunay triangulations. In *ACM SIGGRAPH 2006*, pages 1049–1056, 2006.

[IS00] Martin Isenburg and Jack Snoeyink. Face fixer: Compressing polygonal meshes with properties. In *ACM SIGGRAPH 2000*, pages 263–270, 2000.

[Jac88] G. Jacobson. *Succinct Static Data Structures*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 1988.

[Kar98] George Karypis. *The Metis Homepage*, 1998. http://glaros.dtc.umn.edu/gkhome/views/metis.

[KCS98] L. Kobbelt, S. Campagna, and H.-P. Seidel. A general framework for mesh decimation. In *Graphics Interface '98 Proceedings*, pages 43 – 50, 1998.

[KE00] M. Kraus and T. Ertl. Simplification of nonconvex tetrahedral meshes. *Electronic Proceedings of NSF/DoE Lake Tahoe Workshop for Scientific Visualization*, 2000.

[KE01] Martin Kraus and Thomas Ertl. Cell-projection of cyclic meshes. In *Proceedings of IEEE Visualization '02*, pages 215–222, 2001.

[KG98] R. Klein and S. Gumhold. Data compression of multi resolution surfaces. In *Visualization in Scientific Computing*, pages 13–24, 1998.

[KG00] Zachi Karni and Craig Gotsman. Spectral compression of geometry. In Kurt Akeley, editor, *ACM SIGGRAPH 2000, Computer Graphics Proceedings*, pages 279–286, 2000.

[KK98] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. In *SIAM Journal on Scientific Computing*, volume 20, pages 359–392, 1998.

[KL01] Junho Kim and Seungyong Lee. Truly selective refinement of progressive meshes. In *Proceedings of Graphics Interface 2001*, pages 101–110, Ottawa, Canada, 2001.

[KQE04] M. Kraus, W. Qiao, and D. Ebert. Projecting tetrahedra without rendering artifacts. In *Proceedings of IEEE Visualization '04*, pages 27–34, 2004.

[KSE04] T. Klein, S. Stegmaier, and T. Ertl. Hardware–accelerated reconstruction of polygonal isosurface representations on unstructured grids. In *Proceedings of Pacific Graphics '04*, pages 186–195, 2004.

[KW05] Peter Kipfer and Rüdiger Westermann. Gpu construction and transparent rendering of iso-surfaces. In *Vision, Modelling and Visualization (VMV)*, pages 241–248, 2005.

[LC87] William E. Lorensen and Harvey E. Cline. Marching cubes: a high resolution 3d surface construction algorithm. In *Computer Graphics (Proceedings of SIGGRAPH 87)*, volume 21, pages 163–169, 1987.

[Lev02] Joshua Levenberg. Fast view-dependent level-of-detail rendering using cached geometry. In *IEEE Visualization 02*, pages 259–266, 2002.

[LH04] Frank Losasso and Hugues Hoppe. Geometry clipmaps: Terrain rendering using nested regular grids. In *ACM SIGGRAPH 2004*, pages 769–776, 2004.

[Lin03] Peter Lindstrom. Out-of-core construction and visualization of multiresolution surfaces. In *ACM SIGGRAPH 2003 Symposium on Interactive 3D Graphics*, pages 93–102, 2003.

[LS01] Peter Lindstrom and Claudio T. Silva. A memory insensitive technique for large model simplification. In *IEEE Visualization 2001*, pages 121–126, 2001.

[LT98] Peter Lindstrom and Greg Turk. Fast and memory efficient polygonal simplification. In *IEEE Visualization 98*, pages 279–286, 1998.

[LT99] Peter Lindstrom and Greg Turk. Evaluation of memoryless simplification. *IEEE Transactions on Visualization and Computer Graphics*, 5(2):98–115, April-June 1999.

[LT00] Peter Lindstrom and Greg Turk. Image-driven simplification. *ACM Transactions on Graphics*, 19(3):204–241, July 2000.

[Max95] Nelson Max. Optical models for direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):99–108, June 1995.

[MDM04] S. Marchesin, J.-M. Dischler, and C. Mongenet. 3d roam for scalable volume visualization. In *IEEE Symposium on Volume Visualization and Graphics*, pages 79–86, 2004.

[MGS02] Michael Meissner, Stefan Guthe, and Wolfgang Straßer. Interactive lighting models and pre-integration for volume rendering on pc graphics accelerators. In *Graphics Interface 02*, pages 209–218, 2002.

[MHC90] N. L. Max, P. Hanrahan, and R. Crawfis. Area and volume coherence for efficient visualization of 3d scalar functions. *Computer Graphics (San Diego Workshop on Volume Visualization)*, 24(5):27–33, 1990.

[Mn98] M. Mntyl. *An Introduction to Solid Modeling*. Computer Society Press, 1998.

[MPSS05] O. Mallo, R. Peikert, C. Sigg, and F. Sadlo. Illuminated lines revisited. In *Proceedings of IEEE Visualization '05*, pages 19–26, Oct 2005.

[MR00] J. Ian Munro and Venkatesh Raman. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing*, 31(3):762–776, 2000.

[Nie97] G. M. Nielson. *Tools for triangulations and tetrahedralizations and constructing functions defined over them*, chapter 20, pages 429–525. IEEE Computer Society, Silver Spring, MD, 1997.

[Paj01] Renato Pajarola. Fastmesh: Efficient view-dependent meshing. In *Proceedings Pacific Graphics 2001*, pages 22–30, 2001.

[Pas04] V. Pascucci. Isosurface computation made simple: Hardware acceleration, adaptive refinement and tetrahedral stripping. *IEEE TCVG Symposium on Visualization (VisSym)*, pages 293–300, 2004.

[PH97] Jovan Popovic and Hugues Hoppe. Progressive simplicial complexes. In *Proceedings of ACM SIGGRAPH 1997*, pages 217–224, 1997.

[Pom00] Alex A. Pomeranz. Roam using surface triangle clusters (rustic). Master's thesis, University of California at Davis, 2000.

[Pop03] S. Popinet. Gnu triangulated surfaces (gts) library. http://gts.sourceforge.net, 2003.

[PRS99] Renato Pajarola, Jarek Rossignac, and Andrzej Szymczak. Implant sprays: Compression of progressive tetrahedral mesh connectivity. In *Proceedings of IEEE Visualization 1999*, pages 299–306, 1999.

[Pup96] E. Puppo. Variable resolution terrain surfaces. In *Eight Canadian Conference on Computational Geometry*, pages 202–210, 1996.

[Pup98] E. Puppo. Variable resolution triangulations. *Computational Geometry Theory and Applications*, 11(3-4):202–210, December 1998.

[RB93] Jarek Rossignac and P. Borrel. Multi-resolution 3d approximations for rendering complex scenes. In B. Falcidieno and T. Kunii, editors, *Proceedings of Modeling in Computer Graphics: Methods and Applications*, pages 455–465, 1993.

[RGW+03] Stefan Roettger, Stefan Guthe, Daniel Weiskopf, Thomas Ertl, and Wolfgang Straßer. Smart hardware-accelerated volume rendering. In *Joint Eurographics – IEEE TVCG Symposium on Visualization*, pages 231–238, 2003.

[RKE00] Stefan Roettger, Martin Kraus, and Thomas Ertl. Hardware–accelerated volume and isosurface rendering bases on cell projection. In *IEEE Proceedings Visualization '00*, pages 109–116, 2000.

[RO96] Kevin J. Renze and James H. Oliver. Generalized unstructured decimation. *IEEE Computer Graphics and Applications*, Nov 1996.

[Ros98] Jarek Rossignac. Edgebreaker: Connectivity compression for triangle meshes. Technical Report GIT-GVU-98-35, Georgia Institute of Technology, October 1998.

[RR96] Remi Ronfard and Jarek Rossignac. Full-range approximation of triangulated polyhedra. In *Proceedings of Eurographics '96*, pages 67–76, 1996.

[Rup95] Jim Ruppert. A delaunay refinement algorithm for quality 2-dimensional mesh generation. *Journal of Algorithms*, 18(2):548–585, May 1995.

[SCCB05] C. T. Silva, J. L. D. Comba, S. P. Callahan, and F. F. Bernardon. GPU-based volume rendering of unstructured grids. *Brazilian Journal of Theoretic and Applied Computing (RITA)*, 12(2):9–29, 2005.

[SG98] O. G. Staadt and M. H. Gross. Progressive tetrahedralizations. In *Proceedings of IEEE Visualization '98*, pages 397–402, Oct 1998.

[SG03] R. Sondershaus and S. Gumhold. Meshing of diffusion surfaces for point–based tensor field visualization. In *Proceedings 12th International Meshing Roundtable*, pages 177–188, 2003.

[She96] Jonathan Shewchuk. Triangle: Engineering a 2d quality mesh generator and delaunay triangulator. *Applied Computational Geometry: Towards Geometric Engineering (Lecture Notes in Computer Science*, 1148:203–222, May 1996.

[She02] Jonathan Shewchuk. What is a good linear element? interpolation, conditioning, and quality measures. In *Eleventh International Meshing Roundtable*, pages 115–126, 2002.

[Si05] Hang Si. The tetgen library. http://tetgen.berlios.de, 2005.

[SMW98] C. T. Silva, Joseph S. B. M., and P. Williams. An exact interactive time visibility ordering algorithm for polyhedral cell complexes. In *IEEE Symposium on Volume Visualization*, pages 87–94, 1998.

[SS05a] R. Sondershaus and W. Straßer. Fasttetramesh: Efficient multi resolution tetrahedral meshing. In *Proceedings 13th Conference on Vision, Modeling and Visualization (VMV 05)*, pages 217–224, 2005.

[SS05b] R. Sondershaus and W. Straßer. View-dependent tetrahedral meshing and rendering. In *Proceedings of the ACM Graphite 05*, pages 23–30, 2005.

[SS06a] R. Sondershaus and W. Straßer. Point-based tetrahedral mesh simplification with intersection-free boundary simplification. In *Proceedings 14th Conference on Vision, Modeling and Visualization (VMV 06)*, pages 25–32, 2006.

[SS06b] R. Sondershaus and W. Straßer. Segment-based view dependent tetrahedral meshing and rendering. In *Proceedings of the ACM Graphite 06*, pages 569–576, 2006.

[SS06c] R. Sondershaus and W. Straßer. View dependent rendering of tetrahedral meshes using arbitrary segments. *Journal of the WSCG*, 14:129–137, 2006.

[ST90] P. Shirley and A. Tuchman. A polygonal approximation to direct scalar volume rendering. *ACM Computer Graphics (San Diego Workshop on Volume Visualization)*, 5(4):63–70, 1990.

[SZL92] W. Schroeder, J. A. Zarge, and B. Lorensen. Decimation of triangle meshes. In *Computer Graphics (SIGGRAPH '92 Proceedings)*, number 26, pages 65–70, July 1992.

[THJ99] Issac J. Trotts, Bernd Hamann, and Kenneth I. Joy. Simplification of tetrahedral meshes with error bounds. *IEEE Transactions on Visualization and Computer Graphics*, 5:224–237, Jul 1999.

[THJW98] Isaac J. Trotts, Bernd Hamann, Kenneth I. Joy, and David F. Wiley. Simplification of tetrahedral meshes. In *Proceedings of IEEE Visualization '98*, pages 287–295, Oct 1998.

[UBF+05] D. Uesu, L. Bavoil, S. Fleishman, J. Shepherd, and C. T. Silva. Simplification of unstructured tetrahedral meshes by point sampling. In *Volume Graphics 05*, pages 157–165, 2005.

[VCL+05] Hyu Vo, Steven Callahan, Peter Lindstrom, Valerio Pascucci, and Claudio Silva. Streaming simplification of tetrahedral meshes. Technical report, LLNL technical report UCRL-CONF-208710, 2005.

[WE04] Daniel Weiskopf and Thomas Ertl. Gpu-based 3d texture advection for the visualization of unsteady flow. *Computer Graphics Forum (Eurographics 2004)*, 23:479–488, 2004.

[Wil92] Peter Williams. Visibility ordering meshed polyhedra. *ACM Transactions on Graphics*, 11(2):103–126, 1992.

[WMF02] Brian Wylie, Kenneth Moreland, and Lee Ann Fisk. Tetrahedral projection using vertex shaders. In *IEEE Volume Visualization and Graphics Symposium*, page 712, 2002.

[WPG+97] C. F. Westin, S. Peled, H. Gubjartsson, R. Kikinis, and F. Jolesz. Geometrical diffusion measures for mri from tensor basis analysis. In *Proceedings of ISMRM 97*, pages 147–155, 1997.

[XV96] Julie C. Xia and Amitabh Varshney. Dynamic view-dependent simplification for polygonal models. In Roni Yagel and Gregory M. Nielson, editors, *IEEE Visualization '96*, pages 335–344, 1996.

[YLPM05] S.-E. Yoon, P. Lindstrom, V. Pascucci, and D. Manocha. Cache-oblivious mesh layouts. In *ACM Transactions on Graphics (SIGGRAPH)*, volume 24, pages 886–893, 2005.

[YSGM04] S. Yoon, B. Salomon, R. Gayle, and D. Manocha. Quick-vdr: Interactive view-dependent rendering of massive meshes. In *IEEE Visualization 04*, pages 131–138, 2004.

[ZCK97] Yong Zhou, Baoquan Chen, and Arie Kaufman. Multiresolution tetrahedral framework for visualizing regular volume data. In *IEEE Visualization '97*, pages 135–ff., 1997.

[ZDL02] S. Zhang, C. Demiralp, and D. H. Laidlaw. Visualizing diffusion tensor mr images using streamtubes and streamsurfaces. *IEEE Transactions on Visualization and Computer Graphics*, -:in press, - 2002.

[ZSH96] M. Zöckler, D. Stalling, and H. Hege. Interactive visualization of 3d vector fields using illuminated stream lines. In *Proceedings of the IEEE Visualization '96*, pages 107–113, 1996.

Curriculum Vitae

25. März 1973	geboren in Dresden
09/1979 - 06/1989	Polytechnische Oberschule, Dresden
09/1989 - 06/1991	Erweiterte Oberschule ”Bertolt Brecht”, Dresden Abschluss: Abitur
10/1991 - 09/1994	Studium Technische Informatik an der Berufsakademie Stuttgart Angestellt bei IBM Deutschland, Sindelfingen Abschluss: Diplom-Ingenieur (BA)
10/1994 - 10/1995	Zivildienst Filderklinik, Filderstadt-Bonlanden
10/1995 - 08/2001	Studium Informatik an der Universität Tübingen Abschluss: Diplom-Informatiker
10/2001 - 12/2002	Software-Architekt, MagicMaps GmbH, Tübingen
11/2002 - 12/2006	Wissenschaftlicher Angestellter und Doktorand am Wilhelm-Schickard-Institut für Informatik (WSI), Graphisch-Interaktive Systeme (GRIS), Leiter Herr Prof. Dr.-Ing. Dr. h.c. Wolfgang Straßer Ab 2004 im Sonderforschungsbereich (SFB) 382 der Deutschen Forschungsgemeinschaft (DFG) ”Verfahren und Algorithmen zur Simulation physikalischer Prozesse auf Höchstleistungsrechnern”