Dipartimento di Informatica e
Scienze dell'Informazione

Multi-resolution Modeling of Discrete Scalar Fields

by

Emanuele Danovaro

Theses Series **DISI-TH-2005-01**

DISI, Università di Genova
v. Dodecaneso 35, 16146 Genova, Italy http://www.disi.unige.it/

Università degli Studi di Genova

Dipartimento di Informatica e

Scienze dell'Informazione

Dottorato di Ricerca in Informatica

Ph.D. Thesis in Computer Science

Multi-resolution Modeling of Discrete Scalar Fields

by

Emanuele Danovaro

May, 2005

Dottorato di Ricerca in Informatica
Dipartimento di Informatica e Scienze dell'Informazione
Università degli Studi di Genova

DISI, Univ. di Genova
via Dodecaneso 35
I-16146 Genova, Italy
http://www.disi.unige.it/

Ph.D. Thesis in Computer Science

Submitted by Emanuele Danovaro
DISI, Università di Genova
danovaro@disi.unige.it

Date of submission: March 2005

Title: Multi-resolution Modeling of Discrete Scalar Fields

Advisor: Leila De Floriani
DISI, Università di Genova
deflo@disi.unige.it

Ext. Reviewers: Alberto Paoluzzi
Dipartimento di Informatica e Automazione, Università Roma Tre
paoluzzi@dia.uniroma3.it

Amitabh Varshney
Department of Computer Science and UMIACS, University of Maryland
varshney@cs.umd.edu

Abstract

Thanks to improvement in simulation, high resolution scanning facilities and multidimensional medical imaging, the size of geometrical dataset is rapidly increasing, and huge datasets are commonly available. The purpose of this thesis is dealing with large sets of two- or three-dimensional discrete data sets describing scalar fields, by using a multi-resolution approach. Specifically our aim is in designing and implementing data structures and algorithms for multi-resolution models of scalar fields described by large data sets for scientific data analysis and visualization. To achieve this goal we have identified and developed three tasks, namely, design and development of compact data structures for multi-resolution models which operate in main memory, design and development of out-of-core algorithms for building and querying multi-resolution models, as well as out-of-core data structures for implementing such models, and design and development of multi-resolution models of scalar fields which take into account the underlying field morphology.

In this thesis, we have first developed and experimented compact data structures for multi-resolution models for two-dimensional scalar fields based on triangle meshes and built through iterative vertex insertion/removal. Such representations can be also applied to encode multi-resolution models of free-form triangulated surfaces. We have designed and developed multi-resolution data structures for multi-resolution models for 3D scalar fields, based on vertex insertion/removal, and on half-edge collapse. We have performed an extensive experimental comparisons with a multi-resolution model built through full-edge collapse and a multi-resolution model based on regular nested tetrahedral meshes

Largest datasets exceed main memory even if compact data structures are adopted. For these reasons, we have investigated out-of core approaches for encoding and manipulating multi-resolution models for 3D shapes, two- and three-dimensional scalar fields, independently of the simplification strategy used to generate the LOD model. The major issue here has been investigating and experimenting effective techniques for clustering the modifications forming the multi-resolution model. In our research, we have developed out-of-core algorithms for generating and querying multi-resolution models out-of-core, which can be coupled with different representations for the modifications in the model.

One of the major motivations in using multi-resolution models based on unstructured meshes for describing a scalar field lies in their ability of encompassing morphological information. This is the first step towards development of more powerful multi-resolution models based on the combined representation of the geometry and the morphology of the

field. To this aim, we have developed techniques for extracting morphological information from a scalar field and we have developed an algorithm for a generating a multi-resolution representation of a two-dimensional scalar field which maintains the morphology of the field at different levels of resolution.

To my families

Acknowledgements

First of all I want to thank my advisor, Prof. Leila De Floriani: I started my PhD program as a student, she taught to me how to become a researcher. I also wish to thank to Prof. Hanan Samet for helpful discussion and suggestion, to Prof. Enrico Puppo for its challenging point of view, to Prof. Paola Magillo for her continuous support, and to my external reviewers Prof. Amitabh Varshney and Prof. Alberto Paoluzzi. I also want to thank Laura, Davide, Andrea, Neta, Mostefa, Lidija for sharing their efforts with me.

A very special thank to Maria: much more than a colleague and a wife... you made this thesis possible.

Table of Contents

Chapter 1

Introduction

Modern acquisition devices and computer simulations have produced huge data sets describing (static as well as time-varying) spatial objects, scalar and, vector fields for different applications domains, such as industrial design, medical imaging, entertainment, and Geographical Information Systems (GISs). Examples include terrain models, large scanned data sets of real-word objects or scenes, large models of complex CAD objects, of architectural buildings, of synthetic environments, medical data sets, and sets derived from time-varying simulations of physical phenomena. A model obtained through laser range scanning, or a CAD model, can contain up to hundred of millions triangles (for instance, the model of St. Matthew statue contains 372 million triangles [LPC+00], the Newport News double edge tanker consists of 82 million triangles [EMB01]). Medical data sets, or volume data sets generated by simulations (e.g,, computational fluid dynamics), can be composed of billions of points (for instance, medical data from the National Library of Medicine's Visible Human, or the simulation data from the Department of Energy's ASCI project).

The huge size of available data sets has led to an intense research activity on mesh simplification and on *multi-resolution* (also called *Level-Of-Detail (LOD)*) mesh-based models. Data simplification has become popular in the last few years as a tool for reducing a mesh to a manageable size. On the other hand, simplification is a time-consuming task, which cannot be performed on-line on meshes of large size. A multi-resolution model encodes the steps performed by a simplification process as a partial order, from which a virtually continuous set of meshes at different LODs can be efficiently extracted. Multi-resolution representations describe an object, or a scalar field, as a mesh of simple basic elements (triangles, tetrahedra, cuboids) at different resolutions. LOD representations have been developed for triangle meshes, which describe the boundary of a 3D shape, or a terrain (see [DFKP04, DFMPS02, LRC+02] for a survey). Some work has been recently done on LOD models based on tetrahedral meshes for representing volume data sets (see [DFM03]).

There are several research issues concerning LOD representations for scalar fields. LOD data structures are still inadequate to deal with very large meshes describing volume data sets. Thus, the need arises for compact LOD representations as well as for out-of-core LOD data structures and efficient construction and query algorithms. LOD models are designed to allow efficient compression and adaptive refinement in areas of interest, but, at the current state-of-the-art, they do not take into account the morphological structure of the field, expressed by critical points and characteristic lines and regions. In our view, a hierarchical morphological description of the field should be combined with the multi-resolution geometric description provided by a traditional LOD model, thus resulting in a morphology-based LOD model.

In the finite element and computer graphics literature, there has been a burst of research on tetrahedral LOD models for volume data sets generated through nested tetrahedron refinement, thus producing multi-resolution field representations based hierarchies of regular meshes. Nested tetrahedral meshes are suitable to describe volumetric data at different LODs when the data points are regularly distributed in the 3D space. On the other hand, large unstructured tetrahedral meshes are produced in a variety of applications, which include computational fluid dynamics, thermodynamics, structural mechanics. Recently, research has been focused on visualization techniques that work directly on these kinds of meshes, thus making it possible to avoid re-sampling the field onto a regular grid (see, for instance, [CS97, CS98, FS01]). Re-sampling is often not feasible, because of the large variation of cell size and of the non-convex shape of the domain, and it is sometimes not even desirable since the user may want to zoom on small portions of the mesh, and analyze details and the shape of the tetrahedral elements directly in the original decomposition.

In [DFMP97b], a general LOD model based on d-dimensional simplicial complexes, called a *Multi-Tessellation (MT)*, has been introduced, which is both dimension- and application-independent. The MT has been defined for simplicial meshes in arbitrary dimensions and it is independent of any construction strategy. A dimension-independent representation of the MT, and of algorithms for building it and for querying it has been proposed in [Mag99] and implemented in the MT-Library [Mag00]. The MT library contains also algorithms for building an MT for representing free-form surfaces and two-dimensional scalar fields. This has been the starting point of the work described in this thesis. The dimension-independent data structure in the MT library is very general, but it exhibits a large overhead with respect to encoding a full representation of a scalar field or a free-form surface. We have evaluated, for instance, that an MT for describing a 3D scalar field and generated though a simplification algorithm based on edge collapse requires 3.5 times the space necessary to encode a full-resolution representation of the data set.

The purpose of this thesis is dealing with large sets of two- or three-dimensional discrete data sets describing scalar fields by using a multi-resolution approach. Specifically our aim is in designing and implementing data structures and algorithms for multi-resolution models

of scalar fields described by large data sets for scientific data analysis and visualization.

We have identified the following three tasks to achieve the above goal:

- design and development of optimized/compact data structures for multi-resolution models which operate in main memory (*in-core multi-resolution modeling*),
- design and development of out-of-core algorithms for building and querying multi-resolution models (*out-of-core multi-resolution modeling*),
- design and development of structural multi-resolution models of scalar fields which take into account the morphology of the underlying field (*multi-resolution modeling based on morphology*).

We have faced the first task by focusing on compact representations for multi-resolution models for two- and three-dimensional scalar fields. At the beginning, we have studied the problem of encoding a multi-resolution terrain model and we have developed a compact data structure as well as query and construction algorithms operating on it [DDFMP01a]. Most of the data structures we have developed are for encoding three-dimensional scalar fields [DDFMP01b, DDF02, SDDFM03, DDFMP03]. Such data structures are not only more compact than the dimension-independent data structure defined in [Mag99], but achieve considerable compression compared with representing the data set at full resolution. We have performed an analysis and comparison of such data structures, also with other approaches to simplification and multi-resolution modeling of volume data sets. We have implemented software tools based on such data structures, which will be part of the tool repository of the European Network of Excellence AIM@SHAPE. This will also enrich the functionalities of the MT library making it suitable to handle large-size data sets.

When dealing with very large data sets, out-of-core solutions can be very effective. The LOD representations of a huge data set can exceed the available in-core memory, even using a compact representation, and therefore LOD models should be implemented out-of-core. Out-of-core methods have been developed in the literature for simplification and for view-dependent rendering of triangle meshes [CMRS03a, DP02, ESC00, Hop98b, Lin00, LP01, Lin03, Pri00, SG01, VM02], for out-of-core rendering of unstructured tetrahedral data sets [CS97, CS98, FMSW00, FS01], and for encoding nested tetrahedral meshes [Pas03].

In this thesis, we address the problem of out-of-core LOD modeling in a more general setting, i.e., for mesh-based LOD models for 3D shapes, two- and three-dimensional scalar fields, independently of the simplification strategy used to generate the LOD model, in the spirit of the work on the general-purpose Multi-Tessellation (MT). The major issue here is that the relation which defines the dependencies among the components of a multi-resolution model is a partial order, which cannot be described by a tree, but a Directed Acyclic Graph (DAG). Also, the queries which must be performed on the DAG are not

based on classic graph traversals, but they are specific to multi-resolution modeling. Thus, we devoted a considerable effort in defining, implementing and experimenting a very large number of clustering techniques based on DAG traversal and on spatial partitioning and grouping. We have then designed an algorithm for building the selected out-of-core representation, which will be the other component of an out-of-core tool for multi-resolution modeling. This work has been developed under the co-supervision of Dr. Hanan Samet at the University of Maryland at College Park (U.S.A.).

LOD models have been designed to permit efficient compression and adaptive refinement, and, thus, they encode the modifications performed during an error-driven simplification, but they do not maintain information about the morphology of the scalar field. Encompassing morphology would ensure a more accurate reconstruction at different levels of detail. Moreover, we can also envision describing a scalar field as a subdivision of its domain into characteristic lines, regions and 3D cells, and have a model representing such subdivision at different resolutions. This would possibly lead to a more compact representation than a "classical" LOD model based on a simplicial decomposition of the domain of the field. The third task of this thesis moves in this direction. We have focused on two-dimensional scalar fields, but developed algorithms which are can be easily extended to the three-dimensional case [DFMD02, DDFM03b]. We have used the discrete morphological description of the field produced by such algorithms to guide a simplification algorithm based on vertex removal, which maintains the morphology of the field at each simplification level. Based on this algorithm, we have developed a morphology-guided LOD model, from which adaptive representations of the terrain are extracted that have the same morphology of the full-resolution terrain model [DDFM$^+$03a].

This thesis is organized into seven chapters. Below we give a brief description of the content of each chapter.

Chapter 2 provides some background notions which will be used in the thesis. We briefly review the notion of cell and simplicial complex, and data structures for representing them. We also review some concepts from Morse theory and on the watershed transform, which are at the basis of the techniques for extracting morphological information from discrete data.

Chapter 3 presents the state of the art in several fields related to the subject of this thesis. We provide an overview of simplification techniques for two and three-dimensional scalar fields, focusing on current solutions for simplifying unstructured volume datasets described by a tetrahedral mesh. Then, we briefly review techniques for geometry and connectivity compression of triangle and tetrahedral meshes. A large portion of Chapter 3 is devoted to review state-of-the-art data structures for discrete, progressive and variable-resolution LOD models, both for regular and irregular datasets. Then, we review out-of-core techniques for simplification, compression and multi-resolution modeling. Finally, we describe algorithms for extracting morphological information from discrete scalar fields represented as regular

or irregular meshes. The focus here is on techniques for two-dimensional scalar fields.

In Chapter 4, we define the *Multi-Tessellation (MT)*, a variable-resolution continuous LOD model proposed in [DFPM97]. We present basic definitions and properties, following [DFMP99]. A contribution of this thesis is an in-depth analysis of the fundamental operation performed on a multiresolution model, i.e., selective refinement. We define the primitives which must be efficiently implemented by any data structure encoding an MT, and we present a pseudocode description of such algorithms based on the previously defined primitives. This offer an abstraction layer between the selective refinement procedure, and the underlying data structure describing the multiresolution model. In the last section of Chapter 4, we present a compact multiresolution data structure for a two-dimensional MT built though vertex insertion or removal that we have proposed in [DDFMP01a].

In Chapter 5 we describe, analyze and compare some compact data structures we have developed for mesh-based multi-resolution models generated from unstructured tetrahedral meshes through some specific refinement or coarsening operator. Selecting a specific operator results in a loss of generality, compared to the general framework, but it allows defining data structures which are not only much more compact with respect to an application-independent representation, but achieve a good compression with respect to storing the field representation at full resolution. Specifically, we present a data structure designed for a multiresolution tetrahedral model built through vertex insertion or vertex removal [DDFMP02], and its specialization to multi-resolution models built through half-edge collapse, i.e., by contracting an end to one of its endpoints [DDF02]. Because of the interesting properties of multi-resolution models based on half-edge collapse, we have designed and developed a data structure for multi-resolution tetrahedral meshes built through half-edge collapse [SDDFM03], based on a procedural encoding of the effect of half-edge collapse and on the modification of the view-dependent tree [ESV99]. At the end of this chapter, we present experimental comparisons with a multi-resolution data structure for unstructured tetrahedral meshes described in [CDFM$^+$04], and with a hierarchical model for nested tetrahedral meshes generated through recursive tetrahedron bisection [LDFS01].

In Chapter 6, we address the issue of out-of-core multi-resolution modeling, i.e., of designing out-of-core representations for a multi-resolution model which cannot fit in main memory. In our approach, we have faced the problem in a general way. We have analyzed first the I/O operations performed by the basic selective refinement algorithms and designed and implemented a simulation environment which allows us to evaluate a large number of data structures for encoding the multi-resolution model out-of-core. We have designed and developed more than sixty clustering techniques for the modifications in a multi-resolution model, which take into account their mutual dependency relations and their spatial arrangement. We have then designed an algorithm for building the selected representation out-of-core. Based on our implementation of the data structure and of the query algorithms as well as of the out-of-core algorithm for generating such data structure,

we are currently implementing an out-of-core prototype system for multi-resolution modeling which is somehow independent of the way the single modifications are encoded, and thus will provide a general-purpose tool for out-of-core multi-resolution modeling.

In Chapter 7, we present some results in the direction of morphology-based multi-resolution representations of discrete scalar fields. To this aim, we face the problem of handling huge data sets of terrain or volume data by representing them in a more concise way through a structural description. We have restricted our attention to two-dimensional scalar fields, although the heuristic algorithms that we have designed and implemented can be easily generalized to arbitrary dimensions (and, actually, our implementation is dimension-independent). We give a constructive definition of an approximation of the Morse-Smale complex, defined for C^2-differentiable functions, in the discrete case and we propose region-based heuristics for computing such approximation [DFMD02, DDFM03b]. We compare our approach with existing algorithms for computing an approximated skeleton of the Morse-Smale complex from triangle meshes. We define a simplification algorithm guided by the morphology and develop a multi-resolution representation for a two-dimensional scalar field which maintain the structure of the Morse-Smale complex computed on the full-resolution representation in any adaptive representation extracted from this enriched multi-resolution model [DDFM$^+$03a].

Chapter 2

Background

This introductory chapter aims at providing some background notions which will be used in the remaining of this thesis. In Section 2.1, we introduce the definition of cell and simplicial complexes, which are the basic combinatorial structures on which our work is based. Section 2.2 presents the most common data structures for representing simplicial complex. Section 2.3 reviews the mathematical basis for morphological characterization of scalar fields. It briefly describes Morse theory and the watershed transform in the continuous case.

2.1 Cell and Simplicial Complexes

In this Section, we introduce the definitions of cell and simplicial complexes and their properties. For more details on point sets and algebraic topology, see [Kel55, Mas91].

2.1.1 Cell Complexes

Intuitively, a Euclidean cell complex is a collection of basic elements, called *cells*, which cover a domain in the Euclidean space [Man87]. A cell complex used as a decomposition of the domain of a scalar field, or as decomposition of a surface, is often called a *mesh*.

A *k-dimensional cell* (k-cell) γ in the Euclidean space $\mathbb{E}^n$, $1 \leq k \leq n$, is a subset of $\mathbb{E}^n$ homeomorphic to an open k-dimensional ball $B^k = \{x \in \mathbb{E}^k : ||x|| < 1\}$ (where $||x||$ denotes the norm of vector x). A 0-cell is a point in $\mathbb{E}^n$, k is called the *order*, or the *dimension*, of a k-cell γ.

The *relative boundary* $b(\gamma)$ of a k-cell γ, $1 \leq k \leq n$, is the boundary of γ with respect to

the topology induced by the usual topology of $\mathbb{E}^n$. Note that the relative boundary of a 0-cell is empty. The *combinatorial boundary* $B(\gamma)$ of a cell γ is the collection of cells γ' of Γ such that $\gamma' \subseteq b(\gamma)$ (as a point set).

A *Euclidean cell complex* in $\mathbb{E}^n$ is a finite set Γ of cells of dimension at most d, $0 \leq d \leq n$, such that the cells of Γ are disjoint, and the combinatorial boundary of each k-dimensional cell consists of cells of Γ of dimension less than k. The maximum d of the dimensions of the cells in a complex Γ is called the *dimension*, or the *order*, of the complex.

A *maximal cell* is any d-cell in a complex Γ of dimension d. A cell which is not on the boundary of any other cell of Γ is called a *top cell*. An h-cell γ' which belongs to the combinatorial boundary $B(\gamma)$ of a cell γ is called an *h-face* of γ. If $\gamma' \neq \gamma$, then γ' is called a *proper face* of γ. Note that each cell γ is a face of itself.

The *domain* (or *carrier*) $\Delta\Gamma$ of a Euclidean cell complex Γ is the subset of $\mathbb{E}^n$ spanned by the cells of Γ. A subset Λ of Γ is called a *subcomplex* of Γ if and only if Λ is a cell complex. The *k-skeleton* of a d-dimensional Euclidean cell complex Γ is the subcomplex of Γ which consists of all the cells of Γ of dimension less or equal to k, where $0 \leq k \leq d$.

The *closure* $\bar{\Lambda}$ of a subset Λ of Γ is the smallest subcomplex of Γ which contains Λ. The closure of Λ consists of all the cells of Λ together with all of their faces. The *star* (or *combinatorial co-boundary*) $St(\gamma)$ of a cell in a Euclidean cell complex Γ is the union of $\{\gamma\}$ with the set of all cells γ' of Γ which contain γ in their combinatorial boundary, i.e., $St(\gamma) = \{\gamma\} \cup \{\gamma' \in \Gamma | \gamma \in B(\gamma)\}$. The *link* $Lk(\gamma)$ of a cell γ is the set of cells of Γ forming the combinatorial boundary of the cells in $St(\gamma) - \{\gamma\}$ not containing γ. Note that $Lk(\gamma)$ is a subcomplex of Γ formed by the cells of $St(\gamma)$ not containing γ, i.e., $Lk(\gamma) = \bar{St}(\gamma) - St(\gamma)$.

The *valence* of an d-cell γ in a cell complex Γ is the number of $(d+1)$-cells of Γ that are in the star of γ.

Regular grids are cell complexes in which all cells are hypercubes (squares or cubes in two and three dimensions, respectively) and all d-cells have the same valence respectively. Moreover, we call *nested (regular) meshes* those meshes which are defined by the uniform subdivision of a d-cell into scaled copies of it. Note that the vertices of a regular mesh are a subset of the vertices of a regular grid and that all sub-cells with a given dimension d have the same valence.

2.1.2 Simplicial Complexes

Simplicial complexes are similar to cell complexes, but their cells, called *simplices*, are closed and defined by the convex combination of points in the Euclidean space.

Let k be a non-negative integer. A *k-simplex* (or a *k-dimensional simplex*) σ is the convex

hull of $k+1$ affinely independent points in $\mathbb{E}^n$ (with $k \leq n$), called *vertices* of σ. k is called the *dimension* of σ. A *face* σ of a k-simplex γ, $\sigma \subseteq \gamma$, is an h-simplex ($0 \leq h \leq k$) generated by $h+1$ vertices of γ. Conversely, γ is said to be a *co-face* of σ.

A *simplicial complex* Σ is a collection of simplices of dimension at most d in $\mathbb{E}^n$, $0 \leq d \leq n$, such that:

(i) all simplices spanned by the vertices of a simplex in Σ are also in Σ (i.e., Σ contains all faces of every simplex in Σ);

(ii) if $\sigma_1, \sigma_2 \in \Sigma$ intersect, then $\sigma_1 \cap \sigma_2 \in \Sigma$ (i.e., the intersection of any two simplices in Σ is either empty or a face of both simplices).

A simplicial complex Σ in $\mathbb{E}^n$ is a topological space. Its underlying space is the union of its simplices and the subspace topology is inherited from $\mathbb{E}^n$.

A d-dimensional simplicial complex Σ in which all top simplices are d-simplices is called *regular*, or *uniformly d-dimensional*. Two simplexes are called *k-adjacent* if they share a k-face. Two p-simplexes, $0 < p \leq d$, are said to be *adjacent* if they are $(p-1)$-adjacent. Two vertices (i.e., 0-simplexes) are called *adjacent* if they are both incident at a common 1-simplex. An *h-path* is a sequence of simplexes $(\sigma_i)_{i=0}^k$ such that two consecutive simplexes σ_{i-1} and σ_i in the sequence are h-adjacent, $0 \leq h \leq d-1$. Two simplexes σ and σ^* are *h-connected* if and only if there exists an h-path $(\sigma_i)_{i=0}^k$ such that σ is a face of σ_0 and σ^* is a face of σ_k.

A regular $(d-1)$-connected d-complex in which the star of any $(d-1)$-simplex consists of one or two simplices is called a *(combinatorial) pseudo-manifold* (possibly with boundary).

A subset M of the Euclidean space $\mathbb{R}^n$ is called a *d-manifold* ($d \leq n$) if and only if every point p of M has a neighborhood homeomorphic to the open d-dimensional ball. A subset M of the Euclidean space $\mathbb{R}^n$ is called a *d-manifold with boundary* ($d \leq n$) if and only if every point p of M has a neighborhood homeomorphic either to the open d-dimensional ball, or to the open d-dimensional ball intersected with a hyperplane in $\mathbb{R}^n$.

A pseudo-manifold satisfying the additional property that all its vertices have a link combinatorially equivalent either to the $(d-1)$-dimensional sphere or to the $(d-1)$-dimensional ball is called a *(combinatorial) manifold*. The domain of a Euclidean simplicial complex which is described by a combinatorial manifold is a manifold in $\mathbb{R}^n$.

2.2 Representing Simplicial and Cell Complexes

In this Section, we review the basic elements for describing a data structure for simplicial complexes, that is topological relations, as well as the most common data structures for two-

dimensional and three-dimensional simplicial complexes with a manifold domain. Finally, in Subsection 2.2.4, we briefly review data structures for simplicial complexes in arbitrary dimension and not necessarily restricted to the manifold domain as well as data structures for cell complexes (see also [DFKP04, DFGH04]).

2.2.1 Topological Relations

The data structures for simplicial and cell complexes are defined and classified in terms of the entities they encode plus their mutual topological relations. In this Subsection, we introduce topological relations for cell complexes, even if we will use their mainly for describing simplicial complexes.

Let Γ be a d-complex and let $\gamma \in \Gamma$ be a p-cell, with $0 \leq p \leq d$. For g, q, $0 \leq g, q \leq d$, we define the following *topological relations*:

- for $p > q$, the *boundary relation* $R_(\gamma)$ consists of the set of cells of order q in Γ which are faces of γ;
- for $p < g$, the *co-boundary relation* consists of the set of cells of order g in Γ in the star of γ which are different from γ;
- for $p > 0$, the *adjacency relation* is the set of p-cells in Γ that are $(p-1)$-adjacent to γ;
- the *adjacency relation*, where γ is a vertex, consists of the set of vertices γ' such that there is a 1-cell γ'' in Γ such that $B(\gamma'') = \{\gamma, \gamma'\}$.

Boundary and co-boundary relations are called *incidence relations*.

2.2.2 Data Structures for Triangle Meshes

All the data structures presented in this Section are for triangle meshes. We call a two-dimensional simplicial complex with a manifold domain a *triangle mesh*. There are essentially two classes of data structures for triangle meshes, those in which edges are encoded as primary entities, called *edge-based representations*, and those in which triangles are considered as primary entities, called *triangle-based representations*. In describing and analyzing these data structures, we will only consider structural information, while disregarding geometric information (which consist just of vertex coordinates), and attributes (which are application dependent).

Edge-based representations have been developed for general meshes and can be used for triangle meshes without any change. The *winged-edge data structure*, originally proposed to represent polyhedra with polygonal faces [Bau75], explicitly encodes all entities forming a complex. For each edge e, the data structure maintains the indexes of the two extreme vertices of e, of the two faces bounded by e, and of the four edges that are both adjacent to e and are on the boundary of the two faces bounded by e. Vertices and faces are also encoded: for each vertex v, a pointer to any edge incident in v, and for each face f, a pointer to an edge bounding f is maintained. Links emanating from edges support efficient navigation of the mesh by traversing either the boundary of a face or the star of a vertex, in either counterclockwise or clockwise order. By assuming the size of an index as a unit, the total storage cost (disregarding geometric and attribute information) is about $27n$, where n is the number of vertices of the complex. Note that the number of edges $\|E\| \approx 3n$ and the number of faces $\|F\| \approx 2n$ from Euler formula. A simplified version of the winged-edge data structure is the *Doubly-Connected Edge List (DCEL)* [MP78] where, for each edge, indexes of just two adjacent edges are maintained. This data structure has a lower cost (about $21n$), but it allows traversing the star of vertices only counterclockwise, and the boundary of faces only clockwise.

A widely-used data structure is the *half-edge data structure* [Man87] in which two copies of the same edge are encoded, oriented in opposite directions. For each edge of the complex, the data structure stores two half-edges, and information about a given edge e is split between such two halves. Each half-edge stores:

- the index of the origin of e;
- the index of the face on the left of e;
- two indexes of the edges preceding and following edge e along the boundary of that face;
- the index of its twin half-edge, which is oriented in the opposite direction.

Information attached to vertices and faces are the same as in the winged-edge or in the DCEL data structures. In the implementation discussed in [Sam05], the cost of the half-edge data structure is equal to that of the winged-edge.

In [CKS98] an efficient implementation of the half-edge data structure for triangle meshes is proposed that combines the advantages of the pointer-based and the index-based approaches by grouping the three half-edges belonging to the same triangle in the subsequent entries `E[3i], E[3i+1], E[3i+2]` of a global array. In this way, the next and previous pointers can be replaced by simple index computations *modulo* 3 while the index of the corresponding triangle is obtained through integer division by 3. As a consequence, each half-edge requires only two pointers: one to its origin and one to its opposite half-edge.

In addition each vertex has a pointer to one incident half-edge. The triangle to half-edge pointer can be replaced with an index multiplication by 3. This sums up to a storage cost of $13n$. The data structure supports simple mesh navigation.

The most common triangle-based data structure is the so-called *indexed data structure*: for each triangle t, three indexes to its vertices are maintained (i.e., $R_{20}(t)$). The cost of this data structure (disregarding geometry and attributes) is about $6n$, where n denotes the number of vertices. The indexed data structure has been extended in order to support mesh traversal through edges by storing, for each triangle t, also the indexes to the three triangles adjacent to t [Law77] that is $R_{22}(t)$. The resulting data structure is called the *indexed data structure with adjacencies*. To support an efficient traversal of the mesh passing through vertices, it is convenient to attach to each vertex of the mesh a pointer to one of its incident triangles (i.e., a partial $R_{01} * (v)$ relation). The storage cost of the extended incidence data structure with adjacencies (disregarding geometry and attributes) is $13n$.

The *Corner Table* proposed by Rossignac et al. [RC99, RSS01] has designed for representing triangle meshes of domains homeomorphic to a 2D sphere, but it has been expanded to represent also manifold meshes with handles and holes, triangulated boundaries of non-manifold solids (in which case the object is decomposed into manifold components), meshes that contain only quadrilaterals or a combination of simply-connected polygonal faces with an arbitrary number of sides. The entities in the Corner Table are triangles and vertices. The Corner Table explicitly encodes the R_{20} incidence relation and the R_{22} adjacency relation of each triangle. In the Corner Table, the "corner" index associates a vertex with one triangle that is incident at it. Given a specific way of indexing the corners of each triangle, the Corner Table supports (in optimal time) the operation to retrieve all other corners that are associated with a vertex. However, when given a vertex, it is impossible to access the corners associated to that vertex, nor the triangles incident at it efficiently.

Indexed data structures do not support explicit encoding of attributes for edges. A competitive data structure, which encodes all entities in a triangle meshes, is a simplified version of the *incidence graph* [Ede87], called the *simplified incidence graph* [BDF90]. In a simplified incidence graph, the following information are encoded:

- for each vertex v, a pointer to one edge incident at v (partial $R_{01} * (v)$ relation);
- for each edge e, the indexes of the two extreme vertices of e ($R_{10}(e)$), and of the two triangles sharing e ($R_{12}(e)$);
- for each triangle t, the indexes of the three edges bounding t ($R_{21}(t)$).

The storage cost of this data structure is equal to $19n$. The simplified incidence graph represents vertices, edges and triangles explicitly and uniquely. Traversal operations are

efficient as in edge-based data structures. The simplified incidence graph is also more space-effective than standard edge-based data structures.

2.2.3 Data Structures for Tetrahedral Meshes

We call a three-dimensional simplicial complex with a manifold domain a *tetrahedral mesh*. Data structures for tetrahedral meshes are a direct extension of data structures for triangle meshes (see Subsection 2.2.2). The indexed data structure and the indexed data structure with adjacencies [Nie97, PBCF93], are defined in a completely similar way as in the case of triangle meshes. If n denotes the number of vertices and $\|T\|$ the number of tetrahedra of a tetrahedral mesh M, encoding connectivity information in an indexed data structure requires $4\|T\|$ indexes. The number of tetrahedra $\|T\|$ can be $O(n^2)$ (see [Ede87]). In practice, however, $\|T\|$ has been evaluated to be about $6n$. [CDFM$^+$04]. By assuming the size of an index to be a unit, the total storage cost of the indexed data structure is equal to $24n$ (disregarding geometric and attribute information). Encoding the connectivity in an indexed data structure with adjacencies requires $8\|T\|$ indexes. This leads to a total cost of $48n$, that becomes $49n$, if, for each vertex v, we also encode an index to one of its incident tetrahedra.

A tetrahedral mesh can also be represented as a simplified incidence graph [BDF90]. In such representation, the following information are encoded:

- for each vertex v, a pointer to one edge incident at v (partial $R_{01} * (v)$ relation);
- for each edge e, the indexes of the two extreme vertices of e ($R_{10}(e)$), and of one of the triangles sharing e (partial $R_{12} * (e)$);
- for each triangle t, the indexes of the three edges bounding t ($R_{21}(t)$), and the indexes of the two tetrahedra sharing it ($R_{23}(t)$);
- for each tetrahedron σ, the indexes of the four triangles bounding σ ($R_{32}(\sigma)$).

The cost of the data structure is: $n + 3\|E\| + 5\|F\| + 4\|T\|$ with n number of vertices, $\|E\|$ number of edges, $\|F\|$ number of triangular faces and $\|T\|$ number of tetrahedra. We can also derive: $\|F\| = 2\|T\| = 12n$ and, from Euler formula $n - \|E\| + \|F\| + \|T\| = 1$, $\|E\| = 7n$. Thus the total cost of the data structure (disregarding geometric information and attributes) is equal to $n + 3\|E\| + 5\|F\| + 4\|T\| = 106n$. Thus, it is about twice as the cost of the indexed data structure with adjacencies. The advantage of a simplified incidence graph is representing all simplices in a tetrahedral mesh explicitly and uniquely, which permits associating attribute information with edges and triangles as well.

Data structures designed for encoding three-dimensional cell complexes like the *Face-Edge* [DL89] and the *Handle-Face* data structures [LT97] could also be used to encode tetrahedral meshes. Both the Face-Edge and the Handle-Face data structures describe the three-dimensional cells in a complex implicitly, by encoding the two-manifold complexes that form the boundary of such cells. The space requirements of the Face-Edge and the Handle-Face data structures, when applied to tetrahedral meshes, are equal to $\|T\| + 18\|F\|$ and $78\|T\|$ indexes, respectively, where $\|T\|$ is the number of tetrahedra and $\|F\|$ the number of faces. Thus, their storage costs are definitely higher than those of the indexed data structures, or of the simplified incidence graph.

2.2.4 Data Structures for Cell and Simplicial Complexes

Dimension-independent data structures have been proposed for d-dimensional manifold complexes, which include the *cell-tuple* data structure [Bri89], the *n-G-map* [Lie91] for cell complexes, and the *Indexed data structure with Adjacencies (IA)* for simplicial complexes (also called *winged representation*) [PBCF93]. In the IA data structure, $2(d+1)$ references are needed for each d-simplex in a d-complex. Both the cell-tuple data structure, and the n-G-map, when used to describe a simplicial complex, require $(d+1)!(d+1)$ references for each d-simplex. The IA can describe Euclidean pseudo-manifolds embedded in the d-dimensional space. The n-G-map and the cell-tuple data structures describe a larger sub-class of pseudo-manifolds introduced in [Lie91]. However, none of them can encode arbitrary cell, or simplicial, complexes.

Selective Geometric Complexes (SGCs) [RO90] can describe non-manifold and non-regular objects represented through cell complexes whose cells can be either open, or not simply connected. In SGCs, cells and their mutual adjacencies are encoded in an incidence graph [Ede87]. The *incidence graph* is a data structure for arbitrary cell complexes, which encodes all the cells of the complex, and for each k-cell γ, all the $(k-1)$-cells bounding γ, and all the $(k+1)$-cells in the star of γ. Thus, it provides a complete, but verbose description of the complex.

Data structures for non-manifold, non-regular two-dimensional cell complexes have been proposed for modeling non-manifold solids [GCP90, Wei88, YK95]. Any three-dimensional cell is encoded implicitly through the manifold 2-complex partitioning its boundary. Experimental evaluations reported in [LL01] show that these data structures do not scale well to the manifold case, since their storage cost is between 2.1 and 4.4 times higher than that of the *winged edge* data structure [Bau72]. The *partial entity structure* [LL01] has been shown to require half of the space of the radial-edge structure. A data structure for encoding two-dimensional simplicial complexes, called the *triangle-segment* data structure, has been proposed in [DFKP04], which extends the IA data structure to deal with non-regular parts and with non-manifold adjacencies of two-simplices at an edge.

The triangle-segment data structure is compact, and scales very well to the manifold case, since it requires just one Byte per vertex more than the IA data structure when applied to a manifold complex. In [DFMMP03], a compact data structure for arbitrary three-dimensional simplicial complexes embedded in the three-dimensional Euclidean space is described, called the *Non-Manifold Indexed data structure with Adjacencies (NMIA)*. The NMIA data structure scales very well to the case of simplicial three-dimensional manifold complexes, since it exhibits an overhead of just one Byte per vertex with respect to the IA data structure when applied to manifold complexes. An alternative approach to the design of non-manifold data structures consists of decomposing a non-manifold object into simpler and more manageable parts [DS92, FR92, GTLH98, RC99]. Such techniques deal with the decomposition of the boundary of a regular object into two-manifold parts. In [DFL03], a sound decomposition for d-dimensional non-manifold objects described through simplicial complexes is defined, which is unique and produces a description of a d-complex as a combination of nearly manifold components. A two-level data structure which encodes decomposition as been proposed in [DFH04].

2.3 Morphological Representation of Scalar Fields

The mathematical basis for defining a morphological description of a scalar field is provided by Morse theory [Mil63], and by the watershed transform in the continuous case, that we review in this Section.

2.3.1 Morse Theory

Morse theory is a powerful tool to capture the topological structure of a shape. In fact, Morse theory states that it is possible to construct topological spaces equivalent to a given differential manifold describing the surface as a decomposition into primitive topological cells, through a limited amount of information [GP74, Mil63, Gri76, Bot98, For98, Sma60].

Let f be a C^2-differentiable real-valued function defined over a domain $D \subseteq \mathbb{E}^2$. A point $p \in \mathbb{E}^2$ is a *critical point* of f if and only if the gradient ∇f of f vanishes at p, i.e., if and only if $\nabla f(p) = 0$. Function f is said to be a *Morse function* if and only if all its critical points are non-degenerate, i.e., if and only if the Hessian matrix $Hess_p f$ of the second derivatives of f at p is non-singular (its determinant is $\neq 0$). This implies that the critical points of f are isolated. The number of negative eigenvalues of $Hess_p f$ is called the *index* of a critical point p. In 2D, there are three types of non-degenerate critical points. A non-degenerate critical point p is a *minimum* (*pit*), a *saddle* (*pass*), or a *maximum* (*peak*) if and only if p has index 0, 1 or 2.

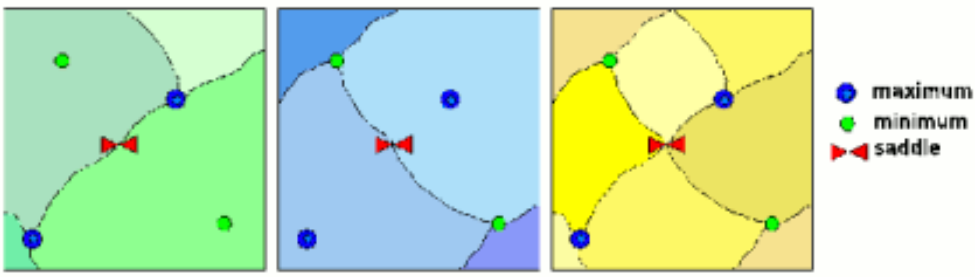


Figure 2.1: (left) An unstable Morse decomposition. (center) A stable Morse decomposition. (right) The corresponding critical net.

An *integral line* of a function f is a maximal path which is everywhere tangent to the gradient vector field ∇f of f. The classical Taylor formula shows that integral lines follow the gradient directions in which the function has the maximum increasing growth. Integral lines cannot be closed, nor infinite (in a compact domain D) and they cover all of D. An integral line is emanating from a critical point or from the boundary of D, and it reaches another critical point, or the boundary of D. An integral line which connects a minimum to a saddle, or a saddle to a maximum is called a *separatrix*. In the GIS literature, separatrix lines that connect minima to saddles are called *course or ravine lines*, while those that connect saddles to maxima are *ridge lines*.

Integral lines that converge to (originate from) a critical point p of index $\imath$ form an $\imath$-cell ((2-$\imath$)-cell) called a *stable (unstable) manifold* of p. The stable (unstable) manifolds are pairwise disjoint and decompose the domain D of f into open cells which form a complex, since the boundary of every cell is the union of lower-dimensional cells. Such complexes are called *stable* and *unstable Morse complexes*.

A Morse function f is called a *Morse-Smale function* if and only if the stable and unstable manifolds intersect only transversely. This means that stable and unstable 1-manifolds cross when they intersect, and the crossing point is a saddle. Cells that are obtained as the intersection of stable and unstable manifolds of a Morse-Smale function f decompose D into a *Morse-Smale complex*. The 1-skeleton of a Morse-Smale complex consists of critical points and separatrix lines. It is called a *critical net*. Figure 2.1(left) gives an example of an unstable complex, where unstable regions of minima are 2-cells, unstable regions of saddles are 1-cells, and unstable regions of maxima are 0-cells (maxima themselves). In Figure 2.1(center), a stable decomposition for the same dataset is illustrated, while Figure 2.1(right) shows the Morse-Smale complex and its critical net.

2.3.2 The Watershed Transform

The watershed transform has been first introduced in image processing for gray-scale images, and several definitions exist in the discrete case [BL79, BM98, MR96, Mey94, SS00, VS91]. The watershed transform has also been defined for C^2-differentiable functions over a connected domain D for which the critical points are isolated, thus, for Morse functions. The watershed of a C^2-differentiable function f is defined based on the concept of topographic distance [Mey94].

Definition 2.3.1 *The* topographic distance $T_f(p,q)$*, with respect to a* C^2*-differentiable function* f *over a domain* D*, between two points* $p, q \in D$ *is defined as*

$$T_f(p,q) = \inf_P \int_P ||\nabla f(P(s))|| ds$$

where the infimum is over all smooth curves P *inside* D*, such that* $P(0) = p$*,* $P(1) = q$*, and* $||\cdot||$ *is the norm of a vector.*

The topographic distance has been defined in this way in order to ensure that the path which minimizes the topographic distance is the path of steepest slope. In other words, if p and q are two points in D, with $f(p) < f(q)$ $(f(q) < f(p))$, and if there is an integral line c which reaches both p and q, then

$$T_f(p,q) = f(q) - f(p) \quad (T_f(p,q) = f(p) - f(q))$$

Otherwise,

$$T_f(p,q) > |f(q) - f(p)|$$

Let $\{m_i\}$, $i \in I$ be the set of minima of f (where I is some index set).

Definition 2.3.2 *The* catchment basin $CB(m_i)$ *of a minimum* m_i *is the set*

$$CB(m_i) = \{p \in D : f(m_i) + T_f(p, m_i) < f(m_j) + T_f(p, m_j), j \in I - \{i\}\}.$$

The *catchment basin* $CB(m_i)$ is formed by the set of points which are closes to m_i than to any other point in P with respect to the topographic distance.

Definition 2.3.3 *The* watershed $WS(f)$ *of* f *is the set of points in* D *which do not belong to any catchment basin, i.e.,*

$$WS(f) = D - \bigcup_{i \in I} CB(m_i).$$

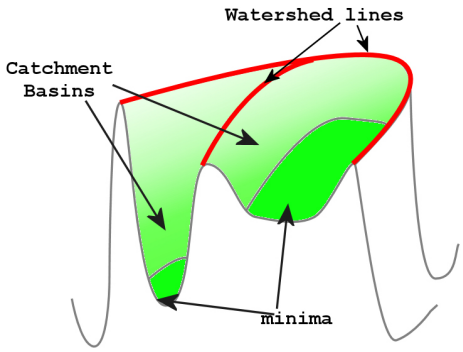


Figure 2.2: The catchment basins of the minima, delineated by watersheds (watershed lines).

In the C^2-differentiable case, it can be easily seen that the catchment basins of the minima of a function f defined by topographic distance are the 2-cells in the unstable Morse-Smale complex of f. If we consider function $g = -f$, then the catchment basins of the minima of function g give the 2-cells of the stable Morse-Smale complex.

Chapter 3

State of the art

This chapter presents the state of the art in several fields related to the subject of this thesis, mainly simplification and compression of two- and three- dimensional scalar fields described by simplicial meshes, in-core multiresolution modeling, simplification and multi-resolution modeling in external memory, and algorithms for morphological analysis of scalar fields. The chapter is organized in five sections.

Section 3.1 provides an overview of simplification techniques for two- and three-dimensional scalar fields. Specifically, we will focus our attention to current solutions for the simplification of an unstructured volume datasets represented by a tetrahedral complex. Section 3.2 reviews techniques for compression of geometric models. We first explain how vertex coordinates may be compressed through quantization, prediction, and entropy coding. Then, we describe how the connectivity of the mesh be compressed. Section 3.3 reviews state-of-the-art data structures for discrete, progressive and continuous multiresolution models, both for regular and irregular datasets. Section 3.4 reviews techniques for simplification, compression and multi-resolution modeling in external memory. Finally, Section 3.5 describes state-of-the-art techniques for extraction of morphological information from a discrete scalar field represented as regular or irregular meshes. The focus is on techniques for two-dimensional scalar fields.

3.1 Simplification of Simplicial Meshes

The two most common methodologies used in mesh simplification are *refinement* and *decimation*. A refinement algorithm is an iterative algorithm which starts with an initial coarse approximation and adds elements at each step. On the contrary, a decimation algorithm starts with the original mesh and iteratively removes elements at each step. Both

refinement and decimation share a very important characteristic: they try to derive an approximation of the dataset through a transformation of some initial mesh approximating it.

3.1.1 Simplification of Triangulated Surfaces

There has been a large body of literature on simplification of triangulated terrain and triangulated free-form surfaces. In this subsection, we give an overview of such techniques focusing on those working on manifold triangulated surfaces (see [Gar99, LRC+02]). The quantitative results are taken from [Gar99]. We focus on techniques which to preserve the topological type of the Surface, or of the underlying domain.

Two-dimensional scalar fields (also called height fields, or terrain) are among the simplest type of surfaces. They can be defined as the set of points satisfying an equation of the form $z = f(x, y)$ defined in a compact domain of the Euclidean plane $\mathbb{R}^2$. Agarwal and Suri[AS94] have shown that computing an L^∞ optimal approximation of a height field is NP-Hard.

Datasets defining a height field are often huge. Thus, we are interested in algorithms that can quickly compute accurate approximations. Refinement is the most popular approach for terrain approximation. One common algorithm begins with a minimal approximation and iteratively inserts the point where the approximation and the original are farthest apart. Incremental Delaunay triangulation [GS85, DFP95] is often used to triangulate the selected vertices, but other data-dependent triangulations can produce approximations with lower error.

Decimation algorithms for simplifying height fields have also been proposed. Depending on the algorithms selected, decimation may produce higher quality results than refinement. The greater speed and smaller memory requirements of refinement made this latter the most common choice.

While refinement has traditionally been the method chosen for approximating height fields, decimation has been much more widely used for simplifying general surfaces. Perhaps the primary difficulty with refinement, in this case, involves constructing the starting coarse approximation. If we limit ourselves to refining through simple subdivision rules, then the initial approximation must necessarily have the same topology as the original model. However, not only this prevents us from simplifying the topology, but it is not always easy to discover the topology of the input surface. In what follows, we briefly review decimation methods.

Vertex clustering methods spatially partition the vertex set into a set of clusters and find a representation for all vertices within the same cluster. They are generally very fast,

and work on arbitrary collections of triangles. Unfortunately, they can often produce relatively poor quality approximations. The simplest clustering method is the uniform vertex clustering algorithm proposed by Rossignac and Borrel[RB93]. The most natural way to extend uniform clustering is to use an adaptive partitioning scheme such as octrees [LE97]. Centering cells around important vertices, rather than merely partitioning space, can also lead to improved approximations. Clustering methods tend to work well if the original model is highly over-sampled and the required degree of simplification is not too high. They also tend to perform better when the surface triangles are smaller than the cell size.

One of the most widely used simplification technique is *vertex decimation* [SZL92]. In each step of the decimation process, a vertex is selected for removal, all the faces incident into that vertex are removed from the model, and the resulting hole is retriangulated. Since this retriangulation requires a projection of the local surface onto a plane, these algorithms are generally limited to manifold surfaces.

More recent methods use more accurate error metrics, like the localized Hausdorff error [CCMS97, KCVS98]. They maintain links between points on the original surface and the corresponding neighborhood on the approximation, and the distances between these points and the associated faces define the approximation error. The algorithm by Schroeder et al. [SZL92] is reasonably efficient, both in time and space, but it seems to have some difficulties in preserving smooth surfaces, while the other vertex decimation algorithms produce higher quality results, but they are substantially slower and consume more space.

The larger class of simplification algorithms is based on *edge collapse*. An edge collapse, denoted $(v_i, v_j) \rightarrow v$, collapses two vertices v_i and v_j connected by an edge into a vertex v. Thus, each of the two triangles incident at edge (v_i, v_j) becomes an edge incident at v and all other triangles incident a v_i or v_j become incident at v.

Most of the edge-collapse algorithms, which have been developed in the literature, follow a simple greedy procedure to select a sequence of edge collapses. Each pair being considered for contraction is assigned a cost. The way in which this cost is determined is the primary difference among algorithms of this type. Generally, it reflects the amount of error introduced into the approximation by the edge collapse. At each iteration, the pair of lowest cost is contracted.

Hoppe's algorithm [Hop96] for constructing progressive meshes is based on minimization of an energy function. One of its primary components is a geometric error term. This algorithm produces some of the highest quality results among currently available methods. However, the price for such quality is a huge running time. Hoppe reported running times of around an hour for a model of about 70,000 faces.

The algorithm developed by Guéziec [Gué95] maintains a tolerance volume around the

approximation such that the original surface is guaranteed to lie within that volume. It is faster than Hoppe's algorithm, and it appears to generate good quality results.

In the algorithm of Ronfard and Rossignac [RR96], each vertex in the approximation has an associated set of planes, and the error at that vertex is defined by the maximum of squared distances to the planes in this set. Ronfard and Rossignac show that localized Hausdorff error bounds can be derived from their metric. The resulting approximations appear to have generally good quality, and the algorithm is fairly efficient compared to the more exact algorithms.

The *Simplification Envelopes* presented by Cohen et al. in [CVM+96] guarantees that all points of an approximation are within a user-specifiable distance ϵ from the original model and that all points of the original model are within a distance ϵ from the approximation. Approximation distance can vary across different portions of a model, generating an adaptive simplification. It can be considered as a general framework, for defining the approximation error, that can be coupled with a large collection of existing simplification algorithms.

The quadric error metric developed by Garland and Heckbert [GH97] also defines error in terms of distances to sets of planes. However, it uses a much more efficient implicit representation of these sets. Each vertex is assigned a single symmetric 4x4 matrix which can measure the sum of squared distances of a point to all the planes in the set. While the quadric metric sacrifices some precision in assessing the approximation error, the resulting algorithm can produce quality approximations very rapidly. For example, only 7 seconds are required to simplify a model containing 70,000 triangles.

The memoryless algorithm developed by Lindstrom and Turk [LT99], unlike most of the other simplification algorithms, makes decisions based purely on the current approximation. The reported results suggest that it can generate good quality meshes, and that it is fairly efficient, especially in memory consumption. By way of comparison, simplifying a 70,000 triangle model requires about 2.5 minutes.

3.1.2 Simplification of Three-Dimensional Scalar Fields

Many of techniques reviewed in Subsection 3.1.1 can be extended to the 3D case, for simplifying volume datasets [CDL+04]. A *volume data set* consists of a set of points V in the three-dimensional Euclidean space $\mathbb{R}^3$, and of a collection of field values associated with the points of V. Here, we consider a decomposition of the domain of a volume dataset into a tetrahedral mesh with vertices at V.

The first simplification technique for a tetrahedral dataset was proposed by Williams in [Wil93]. He suggested choosing a random subset of the vertices of the mesh and re-

triangulating them using a Delaunay triangulation [dBSvKO97]. Simple solutions based on uncontrolled subsampling present a serious drawback, since there is no control over the accuracy of the simplified mesh.

Hamann and Chen [HC94] adopted a refinement strategy for the simplification of tetrahedral meshes with a convex domain. Their method is based on the selection of a set of significant points and their insertion into the convex hull of the domain of the dataset. Significant data points are identified by large absolute curvatures obtained by a local least-square approximation. When a point is inserted into the tetrahedral mesh, local modifications (by face/edge swapping) are performed in order to minimize a local approximation error. This process leads to a data-dependent triangulation.

In [CDFM+94] Cignoni et al. proposed another technique based on the Delaunay refinement strategy. Here, the vertex selection criterion consists of choosing the point causing the largest error with respect to the original scalar field. This technique has been successively extended in [CMPS97] for handling non-convex tetrahedral meshes which can be obtained by deformation of convex meshes.

A refinement-based strategy has also been used by Grosso and Greiner [GLE97, GS98a]. Starting from a coarse tetrahedral mesh covering the domain, a hierarchy of approximations of the volume is created through a sequence of local adaptive mesh refinement steps. A very similar approach, based on selective refinement, but limited to regular datasets, has been presented in [ZCK97]. All techniques based on a refinement strategy share a common problem. The domain of the dataset has to be convex (or at least it has to be defined as a warping of a regular computational grid [CMPS97]). The reason lies in the intrinsic difficulty in fulfilling strict geometric constraints while refining a mesh (from coarse to fine) and using just the vertices of the dataset.

Another approach that can be classified as a refinement technique is the clustering method presented in [CHHH03]. It generates a hierarchical representation of an unstructured volumetric scalar data set. The algorithm starts with a coarse data representation, consisting of a single cluster that contains all the sample points of the data set. This cluster is partitioned into two sub-clusters, which are inserted into a priority queue sorted by error. This procedure is applied iteratively. At each step, the cluster with the highest error is partitioned, and its sub-clusters are placed into the queue. Cluster partitioning is done by using a splitting plane that reduces the error in the sub-clusters the most, that is the plane passing through the cluster center and orthogonal to the vector obtained from Principal Component Analysis (PCA) on all four components of the scalar field (x, y, z position and field value). PCA has been used in the simplification of 3D surfaces [Bro00, SG01] to obtain good splitting planes. The cluster refinement process terminates when a maximum number of iterations have been completed, or when the maximum error in the priority queue is below some user-defined threshold. A binary tree of clusters is built using the natural parent-child relationship defined by this splitting procedure.

Renze and Oliver in [RO96] proposed the first 3D mesh decimation algorithm based on vertex removal. Given a tetrahedral mesh Σ, they evaluate the internal vertices of the mesh for removal, in random order. The re-triangulation of the hole left by the removal of a vertex v is done by building the Delaunay triangulation Σ_v of the vertices adjacent to v, and searching for a subset of the tetrahedra of Σ_v whose $(d-1)$-faces match the faces of S. If such a subset does not exist, the vertex is not removed. The latter condition may very often hold if the original complex is not a Delaunay one. This method neither measures the approximation error introduced in the reduced dataset, nor tries to select the vertex subset in order to minimize the error.

Staadt and Gross [GS98a] introduce various cost functions to drive the edge collapse process and present a technique to check (and prevent) the occurrence of intersections on the boundary of the mesh and inversions of the tetrahedra involved in a collapse action. The approach is based on a sequence of tests that guarantee the construction of a robust and consistent progressive tetrahedralization.

A simplification technique based on iterative edge collapse has also been sketched by Cignoni et al. in [CMPS97]. The simplification process is driven by the geometric error introduced in the simplification of the mesh domain (called the *domain error*) and by the relative error of the scalar field associated with the vertexes of the dataset (called the *field error*).

A technique based on error-prioritized tetrahedra collapse has been proposed by Trotts et. al. [THJW98]. Each tetrahedron is weighted based on a predicted increase in the approximation error that would result after its collapse. The collapse of a tetrahedron is implemented by means of three edge collapses. The algorithm gives an approximate evaluation of the field error introduced at each simplification step (based on the iterative accumulation of local evaluations, following the approach proposed by Bajaj et al. for the simplification of 2D surfaces [BS96]). The mesh degeneration caused by the modification of the (possibly non-convex) mesh boundary and the corresponding error are managed by forcing every edge collapse that involves a boundary vertex to be performed on the boundary vertex, and avoiding the collapse of corner vertices. This approach preserves the boundary in the case of regular datasets, but cannot be used to decimate the boundary of a dataset with a more complex domain (e.g. non-rectilinear or non-convex, as occurs frequently on irregular datasets). This approach has been extended and described in more detailed [THJ99], where the basic edge collapse operation is used instead of tetrahedron collapse. A priority queue is used and maintained to keep the edge to be collapsed sorted according to a minimum error criterion.

In [CCM+00], Cignoni et al. presented a framework for the simplification of a tetrahedral mesh that is based on the edge collapse operation. They distinctly characterize the two different kind of errors that can be introduced during the simplification process: the field error, that is the error introduced by approximating the original field of the dataset with a

coarser representation, and the domain error, that is the error caused by the simplification of the boundary of the complex. They describe various techniques to evaluate these errors with different *combinations* of accuracy and efficiency.

Another approach, called *TetFusion* that focuses its attention mainly on the efficiency of the simplification process has been presented by Chopra and Meyer in [CM02]. This algorithm is based on the atomic operation of iteratively collapsing a tetrahedron to a single point. The boundary of the mesh is not simplified to avoid any geometrical problem, and a bound of the error introduced is given.

3.2 Compression Techniques for Simplicial Meshes

In this Section, we briefly review techniques for compact encoding of triangle and tetrahedral meshes. These are the basis for the compact representations for multiresolution models we propose in Chapters 4 and 5.

When encoding connectivity information in a simplicial mesh, it is necessary to store, without loss of information (*lossless* compression), at least the vertices of each simplex. Moreover, often adjacency information are of interest, and it may be interesting to treat objects that are non-manifold and/or have a non-null genus. Another objective is the progressive transmission of a simplicial mesh, thus providing approximations with increasing accuracy.

Techniques for encoding the connectivity of the triangle mesh can roughly be classified into techniques based on *triangle strips* (see Section 3.2.2), and techniques based on graph encoding (see Section 3.2.3). Some techniques based on *graph encoding* have been extended to tetrahedral meshes (Section 3.2.4). A third important group consists of techniques that offer a progressive encoding of simplicial meshes, such meshes called *progressive meshes*, are a special case of multiresolution models. For this reason, they are reviewed in Subsection 3.3.2.

3.2.1 Geometry Compression

The compression of vertex coordinates usually combines three steps: quantization, prediction, and statistical coding of the residues. Quantization truncates the vertex coordinates to a desired accuracy and maps them into integers that can be represented with a limited number of bits. Usually 12 bits for each coordinate ensure a sufficient geometric fidelity for most applications and most models.

The next, and most crucial, geometry compression step involves using a vertex predictor.

Both the encoder and the decoder use the same predictor. Thus, only the residues between the predicted and the correct coordinates need to be encoded. Since most edges are short with respect to the size of the model, adjacent vertices are, in general, close to each other, and the differences between the coordinates are small. Thus, a new vertex may be predicted by a previously transmitted neighbor [Dee95]. Instead of using a single neighbor, when vertices are transmitted in a vertex spanning tree, a linear combination of the four ancestors in the tree may be used [TR98].

The most popular predictor is based on the parallelogram construction [TG98]. Given a triangle $t = (v_1, v_2, v)$, we can predict the third vertex v' of an adjacent triangle $t' = (v_1, v_2, v')$ with $v' = v_1 + v_2 - v$. The parallelogram prediction rule may sometimes be improved by predicting the angle between t and t' from the angles of previously encountered triangles or from the statistics of the mesh. Residues are usually close to zero. This makes them suitable for statistical compression[Sal00]. In practice, the combination of these steps compresses vertex location data to about 7t bits.

3.2.2 Triangle Strips

It is clearly possible to specify each triangle in a mesh by giving its three vertices, but, if we sort triangles in such a way that two consecutive triangles share an edge, then it is sufficient to give just the new vertex. In general, the problem of generating a sequential triangulation, i.e., a triangulation which can be covered with a simple strip, may have no solution. In [AHMS94], it is shown that there are sets of more than eight vertices in general positions which do not admit a sequential triangulation. Then, it is necessary to subdivide the mesh into several strips and encode each strip in the least possible space. Since the problem of minimizing the number of strips covering a give triangle mesh has been shown to be NP-complete [ESV96], research has been concentrated on heuristics to obtain satisfying, although sub-optimal, results, and on enhancing the encoding of the individual strips.

In a triangle mesh, a vertex is incident into six triangles on average. Even with a perfect striping, each vertex is encoded twice on average. Deering [Dee95] proposes the use of a small vertex buffer. Instead of specifying a vertex for a second time, we can address a vertex already present in the buffer. Bar-Yehuda and Gotsman [BYG96] analyze the problem of setting the buffer size. With a sufficiently large buffer, each vertex can be specified only once. It has been shown that a buffer of size $\lceil 12.27\sqrt{n} \rceil$ is sufficient to encode any triangle mesh with n vertices. Moreover, there are triangle meshes which require a buffer of size at least $\lceil 1.649\sqrt{n} \rceil$. Other methods [ESV96, Cho97] try to enhance the subdivision into strips by decomposing the model into regions and striping each region separately.

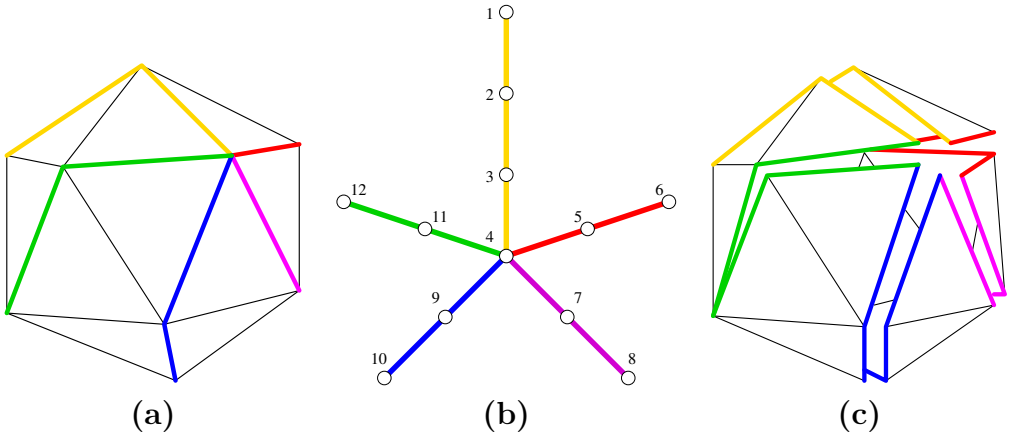


Figure 3.1: (a) A triangle mesh homeomorphic to a sphere, (b) a representation of the *vertex spanning tree*, and (c) the triangle mesh obtained by cutting along edges of the vertex spanning tree

3.2.3 Graph Encoding of Triangle Meshes

More efficient techniques are inspired by a method for encoding planar graphs proposed by Turan [Tur84]. It is important to notice that the connectivity structure of a triangle mesh of null genus is homeomorphic to a planar graph. Thus, graph encoding is equivalent to encoding connectivity information for a triangle mesh. Turan [Tur84] noted that the connectivity of a planar graph with triangular faces can be recovered from the structure of its *vertex spanning tree* and the corresponding *triangle spanning tree*. Given a mesh Σ having V as vertices, a vertex spanning tree is a tree having V as vertices and a subset of edges of Σ as arcs. Figure 3.1 shows a triangle mesh corresponding to a sphere (a), and the corresponding vertex spanning tree (b). After cutting the mesh along edges that are arcs of the vertex spanning tree (see Figure 3.1(c)), we get a triangle spanning tree which is composed by branching triangles and triangle strips (see Figure 3.2). Turan proposed to encode them using a total of roughly $12n$ bits for a graph with n nodes.

Topological Surgery

Topological surgery, proposed by Taubin and Rossignac in [TR98], reduces the cost of the encoding proposed by Turan to $6n$ bits by observing two fact. The binary vertex tree can be encoded with $4n$ bits, using two bits to indicate the presence or absence of left or right child of a node. The corresponding (dual) triangle spanning tree can be encoded with $2n$ bits, one bit per vertex indicating whether the node is a leaf and one indicating whether it is the last child of its parent. Encoding the triangle spanning tree is further improved by applying run-length encoding. An IBM implementation of the Topological Surgery compression has

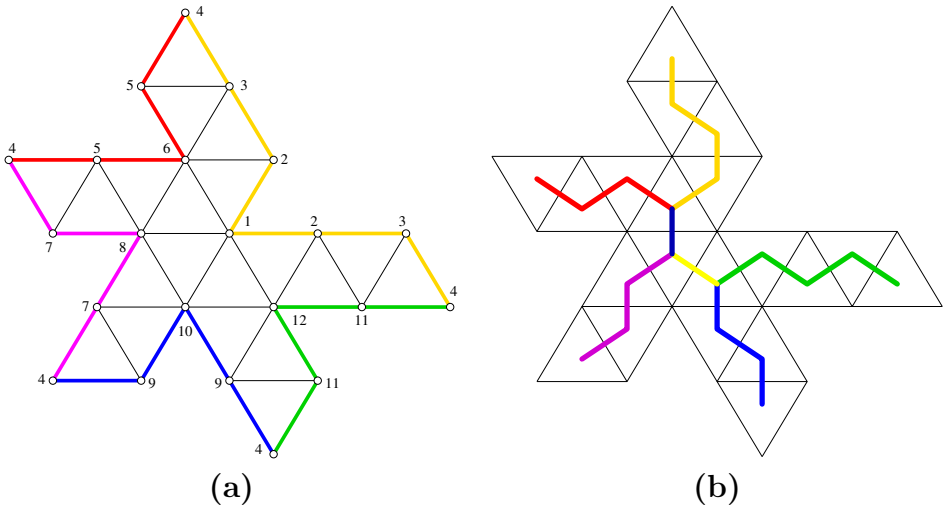


Figure 3.2: (a) Triangle mesh obtained after cutting, and (b) the corresponding triangle spanning tree.

been adopted in the VRML standard [TLHR98] resulting in a 50-to-1 compression ratio with respect to VRML ASCII format. The Topological Surgery approach has been selected as the core of three-dimensional mesh coding (3DMC) algorithm in MPEG-4.

Encoding a Graph Traversal

Several techniques are based on the idea of encoding a traversal of the triangle mesh. Such techniques offer high compression ratios and they implicit encode adjacency information. Gumhold and Strasser [GS98b] and, independently, Rossignac [Ros99] have developed two of the most efficient techniques for triangle mesh compression. Such techniques have strong similarities, since both of them perform a traversal of the triangle mesh and associate a code that reflects the specific configuration of the visited triangle. The sequence of triangle codes can be further compactly encoded.

Edgebreaker [Ros99] has been further improved [KR99, RS99, IS01] and the current version guarantees that a mesh connectivity can be encoded in as low as 1.776 bits per triangle, which is really close to the proven 1.62 bits per triangle theoretical lower bound for encoding planar triangular graphs, proven by Tutte [Tut62]. In certain circumstances, like in semi-regular meshes, these results can be strongly improved and the connectivity cost can be lower than a bit per triangle. A semi-regular triangle mesh is a mesh in which the valence of most of its vertices is constant. Coors and Rossignac [CR04] in the Delphi system have proposed to predict the connectivity of the next triangle. Experiments indicate that, up to 97% of Delphi's guesses are correct, compressing the connectivity down to 0.19 bits per triangle.

Cut Border Machine [GS98b] at first reported connectivity compression results from 1.7 to 2.5 bits per triangle. A context-based arithmetic coder further improves them to 0.95 bits per triangle [GGS99]. In [Gum00], Gumhold proposes a custom variable-length scheme that guarantees less than 0.94 bits for triangle for encoding the most space-consuming case, thus proving that the cut-border machine algorithm has linear complexity.

Encoding Vertex Valences

As a consequence of Euler's theorem applied for triangle mesh we have that the average valence of a vertex is equal to 6. In most available models, the valence distribution is highly concentrated around 6. To exploit this fact, Touma and Gotsman [TG98] have developed a valance-based encoding, which visits the triangles and associate with each vertex its valence plus some extra information. About 80% of the encoding cost lies in the valence, which has a low entropy for nearly regular meshes. This technique cannot guarantee high compression rates in the worst cases, but its average cost is about one.

Alliez and Desbrun [AD01b] have managed to significantly improve the worst case, by developing a heuristic that avoids most expensive configurations. They also show that, if such configurations could be eliminated, the valence-based approach would guarantee to compress the mesh with less than 1.62 bits per triangle.

3.2.4 Compression of Tetrahedral Meshes

Two of the most efficient techniques for triangles meshes have been extended for compact encoding of tetrahedral meshes. In this subsection, we just briefly review techniques. Progressive techniques for tetrahedral meshes are reviewed in Subsection 3.3.2.

Grow & Fold [SR99] presented by Rossignac et al. is an extension of Topological Surgery technique to the case of three dimensional datasets. It encodes a tetrahedron spanning tree (with three bits per tetrahedron) and some additional folding information, to reconstruct the connectivity information of the original mesh. The total cost for encoding mesh connectivity is about 7 bits per tetrahedron.

Gumhold, Guthe and Strasser [GGS99] propose a technique for encoding the connectivity information in a tetrahedral mesh based on a traversal of the mesh. The proposed approach is an extension to three dimensions of the Cut Border method [GS98b]. The number of configurations here is higher, and the cost of the encoding is about 2 bits per tetrahedron on average.

3.3 Mesh-based Multiresolution Models

A mesh-based *multiresolution* (also called *Level-of-Details* (LOD)) model can be seen as a *black box* that is built off-line through a simplification algorithm and is queried on-line for obtaining adaptive meshes on-the-fly. LOD models can be classified into *discrete* and *continuous* LOD models.

In this Section, we review data structures for describing such models. Subsection 3.3.1 describes discrete LOD models, A subclass of continuous models, namely progressive model is described in Subsection 3.3.2. Subsection 3.3.3 reviews continuous variable-resolution models. Specifically, Subsection 3.3.3.1 describes data structures for LOD models based on regular meshes, while Subsection 3.3.3.2 presents data structures for LOD models based on irregular meshes. LOD models for tetrahedral datasets are reviewed in Subsection 3.3.4. An extensive treatment of LOD models for view-dependent visualization and of their applications can be found in [LRC+02, CDL+04], and in [DDFMP03].

3.3.1 Discrete LOD Models

Discrete mesh-based LOD models have been presented for the first time in the seminal work by Clark [Cla76]. Later on, they have been implemented in several commercial products and standards for computer graphics, among which it is worth mentioning VRML and OpenInventor. The underlying idea is very simple. A collection of different representations of an object are computed off-line and stored. An application may select on-line the representation that best suits its needs (in terms of either accuracy, or space and time constraints). The various representations can be obtained easily by running any simplification algorithm several times with different parameters. A sequence of meshes $[\Sigma_0, \ldots, \Sigma_l]$ is obtained, in which each mesh Σ_i is larger and more detailed than those meshes preceding it in the sequence. Each mesh is generally stored independently, together with data about its size and accuracy. The sequence of meshes can be treated as a sorted array of meshes, from which it easy to select the appropriate mesh for an application, on the basis of either accuracy or size.

Note that, for practical purposes, the number of different meshes in a discrete LOD model must be small (usually about five and hardly more than ten). The reduced number of meshes implies a relevant difference in detail between two consecutive meshes in the sequence. When LOD models are used in a dynamic application, this difference may often cause unpleasant "popping" effects. In other words, discrete LOD models do not provide any mechanism for smooth transition between different levels of detail.

In application contexts besides computer graphics, discrete LOD models have been used to manipulate and query geometric models. The purpose of LOD is again to trade-off between

accuracy and complexity, but this time computation needs to focus on specific areas (e.g., for tasks such as point location, spatial selection, collision detection, object intersection, etc.). In this case, data structures, which support navigation of a discrete LOD model, both within a single mesh and across different meshes can be useful [DF89].

LOD models with many levels can be obtained by means of compact data structures that avoid replicating triangles that appear at more than one level [BDFM95, CCMS97]. These models represent a sort of transition from discrete LOD to progressive models.

3.3.2 Progressive LOD Models

Transmission of large datasets through the Web requires long waiting times, even with compact encoding. For such reason, some techniques have been developed which encode a triangle, or a tetrahedral, mesh in a progressive way. A coarse representation is sent first, followed by a sequence of refinement operations. Progressive models are continuous LOD models, since they have a fine granularity, but do not allow extracting meshes in which the resolution varies in different parts of the domain.

Hoppe [Hop96] developed Progressive Meshes (PM) which encode and transmit sequence of vertex split operations one by one. This offers fine granularity refinements, but the encoding is quite expensive. Other techniques, which encode several elementary modifications in a single refinement step, offer high compression ratios with a loss in the granularity of the progressive representation process. Modifications are grouped into 6 to 12 batches and compressed. A transmission cost of 3.5 bits per triangle has been achieved by using one bit per vertex to mark which vertices must be split in each batch [Hop98a, PR00]. Marking the edges, instead of the vertices [TGHL98], allows recovering for free the connectivity of portions of the mesh in which the vertices have a valence equal to 6 and, in general, reduces the cost of the progressive transmission of connectivity to between $1.0t$ bits and $2.5t$ bits. In [AD01a], a method is proposed, which consists of several simplification steps, each of which divides by three the number of vertices. Such methods allow encoding modifications in connectivity with about 1.9 bit per triangles.

The above techniques encode a simplified version of the original connectivity graph and optionally the data necessary to fully restore it to its original form. When there is no need to preserve that connectivity, one may achieve better compression by producing a more regular sampling of the original mesh at the desired accuracy, so as to reduce the cost of connectivity compression, improve the accuracy of vertex prediction, and reduce the cost of encoding the residues.

Popovic and Hoppe [PH97] have proposed a technique for progressive encoding of simplicial complexes of arbitrary dimension, called *Progressive Simplicial Complexes*. Similarly to Progressive Meshes [Hop96], a d-dimensional simplicial complex is encoded through an

approximated complex, usually a single point, and a sequence of refinement steps based on vertex split. This technique is really powerful but each refinement step is quite expensive. It is not practical for mesh compression.

Pajarola, Rossignac and Szymczak [PRS99] propose a technique for the progressive encoding of tetrahedral meshes, called *Implant Sprays*. It can be considered an extension in 3D of Compressed Progressive Meshes (CPMs) [PR00]. It applies at each step a set of vertex split (usually 1/16 to 1/12 of vertices are splitted at each step). Experimental results report an average cost of 3 bits per tetrahedron.

3.3.3 Variable-resolution LOD Models

Variable-resolution LOD models belong to the class of continuous models, but they improve over progressive ones by fully supporting *selective refinement*, i.e., the extraction of meshes with an LOD that can be variable in different parts of the mesh, and can be changed on a virtually continuous scale. This improvement is achieved by observing that modifications that compose an LOD model should not necessarily be organized in a linear sequence, but their dependency relation is actually a partial order. This permits to apply modifications selectively in areas of interest.

3.3.3.1 Regular LOD Representations

LOD models for regular meshes are generated by the recursive application of some basic refinement operator to a regular base mesh, which preserves the regularity of the mesh. When such operators are applied to an irregular base mesh, then semi-regular meshes are obtained. A semiregular mesh is piecewise regular and has only some isolated irregular vertices corresponding to the original base mesh. As refinement proceeds, most parts of the mesh tend to become regular, while some irregularities remain in the proximity of vertices of the base mesh, which may have an arbitrary valence (extraordinary vertices). Basic refinement operators consist of splitting each existing triangle into a set of smaller triangles, according to some predefined pattern. Therefore, recursive refinement generates a tree. Different refinement patterns generate trees with different characteristics.

The simplest and most common refinement operator is *quadrisection*, in which a new vertex is inserted at the midpoint of each edge and each original triangle is split into four subtriangles. The resulting hierarchy is called a *triangle quadtree* [Sam05]. If this operator is applied to a regular mesh where all vertices have degree equal to six (except possibly vertices at the boundary), then the mesh at each level of refinement preserves the same regular structure.

This model has been widely used in the context of finite element methods and for subdivi-

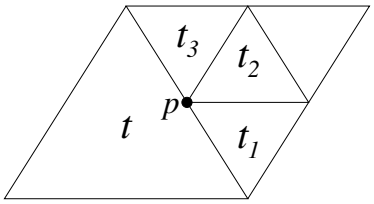


Figure 3.3: A non conforming mesh: triangles t_1, t_2 and t_3 intersect triangle t not at a common simplex.

sion surfaces. Data structures for encoding it are straightforward adaptations of quadtree data structures. Topological operations, such as neighbor finding, can be performed in worst-case constant time by using location codes and bit manipulation [LS00]. From this simple model we cannot extract a simplicial complex, other than those having all triangles at the same level of the hierarchy.

Note that in the finite element and computer graphics literature, a simplicial complex is called a *conforming mesh* (what we call here a mesh). Collection of simplexes such that the boundary of the maximal simplices may not properly intersect are called *non-conforming meshes*. As example of a non-conforming mesh obtained by triangle quadrisection is shown in Figure 3.3, where triangles t_1, t_2 and t_3 intersect triangle t not at a common simplex. For instance $t \cap t_2 = \{p\}$ which belongs to t_2, but not to t.

If a mesh is extracted from a triangle quadtree by considering triangles from different levels of the hierarchy, then the result is always a non-conforming mesh. In order to convert it into a simplicial complex, a post-processing step is usually applied. Some triangles are possibly refined further in order to have a subdivision where two adjacent triangles do not differ for more than one level. Each triangle that is adjacent to some triangles at the next level of the hierarchy is split according to a bisection rule to make it conforming to its adjacent triangles. Bisections introduced in the latter step are undone once further refinement is required [BSW83]. A triangle quadtree, enriched with post-processing bisection, is often known as a *red-green triangulation*, and can support selective refinement. However, selective refinement could not be performed by a simple traversal of the hierarchy. Post-processing to perform bisection requires to traverse the output mesh and to perform neighbor finding operations.

An alternative refinement operator introduces one new vertex per triangle and splits the triangle into three sub-triangles, leading to a situation where all new vertices have degree three, while the degrees of the original vertices are doubled. If used alone, this refinement pattern produces long slivers and never refines the (long) edges of the base mesh. A much better result is obtained if such a refinement operator is interleaved with an operator that flips all "old" edges. Edge flip rebalances the degree of vertices and improves the shape of triangles. After edge flip, all vertices introduced at the previous level will have valence six

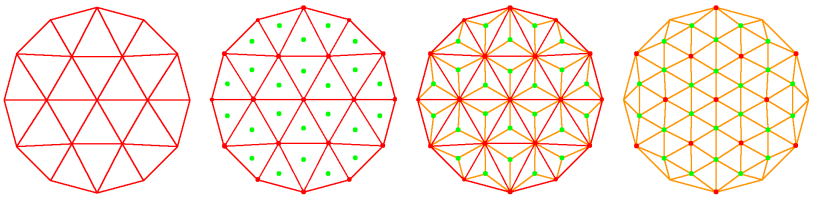


Figure 3.4: Mesh refinement by the $\sqrt{3}$-operator.

and all other vertices will keep their original valence (see Figure 3.4). This operator is called $\sqrt{3}$ *subdivision*, since the double application (one trisection plus one edge flip) produces a mesh that corresponds to a uniform ternary refinement of the original [Kob00]. This model is quite efficient in supporting selective refinement and guarantees that conforming meshes are always extracted.

Another common refinement method based on recursive *triangle bisection* has been adopted by several authors [DWS+97, EKT01, Ger03, LKR+96, Paj98]. This is used especially for terrain data and, more generally, for two-dimensional scalar fields, where data are sampled at the vertices of a regular grid. Extension to a spherical domain is also straightforward, while extension to surfaces of higher genus does not seem not easy. A square universe S is initially subdivided into two right triangles. The bisection rule subdivides a triangle into two similar triangles by splitting it at the midpoint of its longest edge. The recursive application of this splitting rule to S defines a binary tree of right triangles, in which the children of a triangle t are the two triangles obtained by splitting t. Such binary tree is often called a *triangle bintree*.

In order to extract conforming meshes, every new vertex must be inserted into two adjacent triangles at the same time (except for vertices at the boundary). Any pair of triangles which need to be split at the same time forms a *diamond*. Each diamond d is split into four triangles (as shown in Figure 3.5) by the vertex v, called the *split vertex* of d, which bisects the edge shared by the two triangles forming d. The four triangles splitting a diamond form a *split set*. This model is very powerful in supporting selective refinement.

If we consider the split vertex of each diamond, a dual hierarchy of vertex dependencies can also be derived, which can be described by a Directed Acyclic Graph (DAG), in which each vertex v has exactly two parents (namely, those two vertices that generate the parents of triangles forming the split set centered at v) and exactly four children (namely, those four vertices that split triangles forming the split set centered at v). Vertices at the boundary have only two children. An example of such DAG-based representation is depicted in Figure 3.6.

The data structures for representing nested meshes produced through triangle bisections either encode the DAG of the vertex dependencies, or encode a forest of triangles which

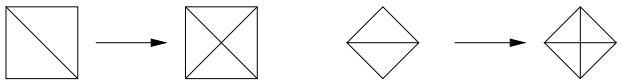


Figure 3.5: Split sets corresponding to the two types of diamonds generated by recursive triangle bisection

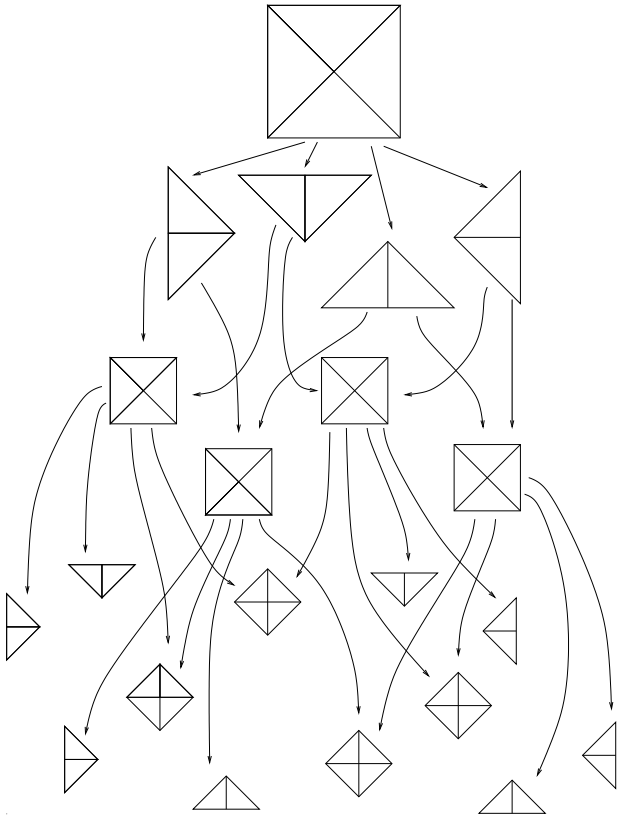


Figure 3.6: The DAG-based vertex dependencies, where each node represents both a vertex and its corresponding diamond (diamonds on the border contain just two triangles).

describes the nested structure of the meshes. The DAG of vertex dependencies can be traversed easily to perform selective refinement while guaranteeing that the result is a conforming mesh [LKR+96, Paj98]. The binary forest of triangles can also be traversed to perform selective refinement, but neighbor finding is necessary to guarantee that when a triangle t is split, also the triangle adjacent to t along its longest edge is split at the same time. In [OR97] a mechanism is proposed, which is based on error saturation, that can extract conforming meshes by traversing the binary forest, without neighbor finding.

If the vertices are available at all nodes of the supporting regular grid, then the forest of triangles is complete, and it can be represented implicitly [DWS+97, EKT01]. Each triangle can be assigned a location code, with a mechanism similar to that adopted for linear encoding of triangle quadtrees [LS00]. Location codes are not stored, but they are used for neighbor finding as well as to retrieve the vertices of a triangle, the value of the field associated with a vertex, etc. An efficient implementation involving arithmetic manipulation and a few bit operations allows performing such computations in constant time [EKT01]. On the contrary, if the binary forest of triangles [the grid of vertices] is not complete, then the forest [the DAG] must be represented explicitly, thus resulting in a more verbose data structure [Ger03]. In [Ger03, LP02], out-of-core implementations are proposed, the former oriented to representing non-complete hierarchies in terrain modeling, the latter targeted at visualization and view-dependent refinement of large terrain data sets.

A variant of the triangle bisection scheme, called a *hierarchical 4-k mesh*, has been proposed in [VdFG99, VG00] in the context of subdivision surfaces. A hierarchical 4-k mesh is based on a combination of vertex insertion on an edge and on edge swap, and it can be applied to an irregular base mesh, thus generating a semi-regular LOD model. This model also can be described by a DAG having nodes with exactly two parents, and either two or four children. This provides a model with characteristics similar to those of the triangle bintrees, with the great advantage of being applicable to surfaces of arbitrary genus.

3.3.3.2 Irregular LOD Representations

Some nested models that refine irregular meshes have been proposed in the literature in connection with the representation of terrain data sets. The model proposed in [SP92] uses a subdivision rule which refines a triangle t by inserting up to four vertices: one in the interior of t and one on each of the three edges of t. Depending on the combination of the vertices inserted, we obtain different predefined subdivision patterns. The model in [DFP95] refines a triangle t by inserting an arbitrary number of vertices lying either inside t or on its edges, and by then computing a Delaunay triangulation. Such models are built according to a refinement strategy, by inserting vertices on the basis of an approximation error. This policy ensures that the same points are inserted on the common face of any two adjacent triangles. Therefore, the resulting LOD mesh admits the extraction of conforming

meshes. Triangles that are refined by inserting vertices on their edges must be expanded together with their neighbors along those edges.

Most models proposed in the literature for irregular meshes, however, are based on non-nested hierarchies. These are direct extensions of progressive models, being built on the basis of simplification algorithms based on local modifications (see Section 3.1).

In [DFMP97a, DFMP98] a general model has been proposed for variable-resolution representation of irregular meshes in arbitrary dimensions, by extending an LOD model for triangle meshes described in [Pup98]. Such general model, called a *Multi-Tessellation* (MT), will be reviewed in details in Chapter 4, since it is at the base of our work. The two-dimensional instance of the Multi-Tessellation, called a *Multi-Triangulation* [DFMP98] has been applied to terrain modeling. The Multi-Triangulation will be described also in Chapter 4. Generally speaking, triangular LOD models consist of a coarse representation of the surface, called a *base mesh*, a collection of local *modifications*, also called *updates*, generated through a refinement or decimation strategy, and a *dependency relation* among modifications which has been proven to be a partial order relation. Data structures for triangular LOD models implement special instance of this framework, as shown in [DFM02]. Also the regular LOD models we have reviewed in Subsection 3.3.3.1, can be viewed as specific instances of such framework.

Compact data structures for LOD mesh-based surface models can be classified into data structures based on *vertex insertion/removal*, and data structures based on *vertex split/edge collapse*.

Data structures based on vertex insertion/removal. *Vertex insertion* in a mesh Σ consists of deleting a connected set of triangles from Σ, which defines the *region of influence* of the vertex v to be inserted, and replacing it with a set of new triangles all incident at v (see Figure 3.7a). The region of influence is defined by the specific mesh refinement algorithm (a technique often based on incremental Delaunay triangulation). *Vertex removal* consists of removing a vertex v together with all the triangles in the star of v and re-triangulating the resulting star-shaped polygon, called the *influence polygon* of v (see Figure 3.7b).

Encoding a vertex insertion requires storing the vertex v to be inserted as well as the subdivision of the region of influence into triangles. This provides also an encoding of vertex removal, since we need to know how to re-triangulate the influence polygon after deleting a vertex v and all the triangles in the star of v.

An implicit encoding for vertex insertion/removal and a compact encoding of the dependency relation as a DAG are described in [KG98]. The DAG encoding is based on connecting the modifications, on which a given modification u depends, in a loop containing u plus all its direct ancestors. Thus, an update u with j direct descendants belongs to $j+1$

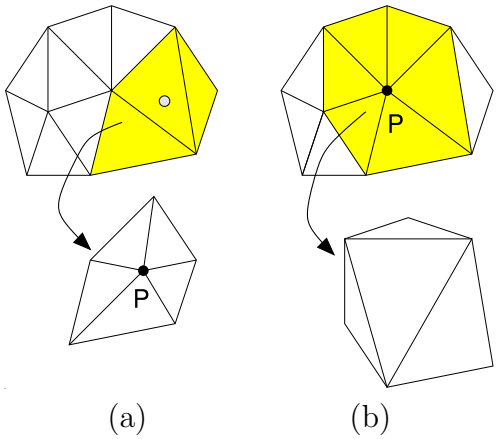


Figure 3.7: (a) Vertex insertion; (b) Vertex removal.

loops: the one associated with u plus those associated with its direct descendants. The total number of links to describe the DAG is equal to $a + h$, where a and h denote the number of arcs and of nodes in the DAG, respectively.

The region of influence of a vertex is encoded by storing a table with all the possible triangulations (up to rotations) of a polygon with s edges (for instance, for $s = 10$, there are 7147 equivalence classes), and identifying the triangulation of the influence polygon by specifying an index in the table. The length of the index is equal to 13 bits, by assuming that there are at most 10 triangles incident at a vertex in any update. The table with the equivalence classes must be stored as well.

In Chapter 4, we will discuss compact encoding of an LOD model based on vertex insertion and provide a comparison with the method in [KG98].

Data structures based on edge collapse/vertex split. LOD data structures based on edge collapse/vertex split extend progressive meshes, described in Subsection 3.3.2.

A naive approach to extracting variable-resolution meshes from a progressive mesh would consist in scanning the sequence of vertex splits and performing only those updates that are necessary to achieve the desired LOD according to the refinement criterion, while skipping all others [Hop96]. However, a vertex split cannot be performed unless the corresponding vertex actually belongs to the current mesh. If a split is skip, this prevents splitting the vertices it generates (which could be considered for splitting at a later stage) and all their descendants. This may lead to an under-refinement of the mesh.

To overcome the above problem, a dependency among vertices, which is represented through a binary forest of vertices, is defined in [Hop97, XESV97]. Roots of the forest are vertices of Σ_0 and the two children of a vertex v are the vertices, v' and v'', generated

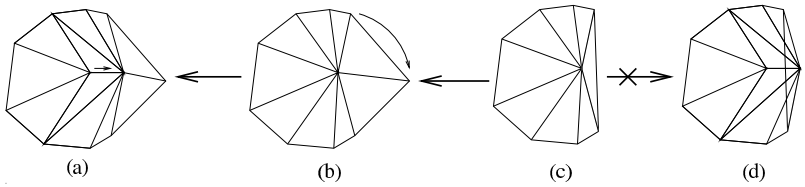


Figure 3.8: Foldover: original sequence of collapses is (a)-(b)-(c) and the reverse sequence of splits is correct; if the central vertex is split as in sequence (c)-(d) a foldover occurs.

by splitting v. Each node in the forest contains all information necessary to perform a vertex split, together with information that subsume the content of its subtree (such as the bounding sphere of the region spanned by all vertices in the subtree, and the maximum approximation error for such vertices). Such information are used to decide whether a certain split must be performed.

In [KL01], a technique is presented which allows re-ordering the vertex splits arbitrarily while still guaranteeing a proper mesh connectivity which corresponds to the original PM connectivity whenever a complete prefix of the vertex splits is executed. However, this simple data structure does not guarantee that a split will be performed in the same situation, i.e., the vertex to be split might not have the same star of triangles as when it was created in the original simplification sequence. This may cause undesirable fold-overs in the mesh (see Figure 3.8). The solution is to encode further dependencies for a vertex v so that v can be safely split. Vertex v will depend on a set of vertices bounding the triangles incident at v produced by the edge collapse which generated v in the simplification sequence [Hop97, XESV97]. In [DFMPS02], it has been shown that in some cases such additional dependencies can be either insufficient to prevent foldovers [Hop97], or redundant [XESV97].

In [ESV99], an effective mechanism based on vertex numbering is introduced, which guarantees that all splits and collapses identified during selective refinement will be performed in the same situation as in the original simplification sequence, thus avoiding foldovers. Vertices are assigned integer labels during simplification. In order to check whether or not a split can be performed, it is sufficient to compare the label of the vertex to be split labels of its adjacent vertices. This is also sufficient to propagate necessary splits/collapses through the hierarchy, in order to support dynamic adjustment of the selectively refined LOD. The resulting data structure, called a *view-dependent tree*, is very compact. Each internal node of the forest corresponds to a vertex created through edge collapse and contains information to perform the corresponding vertex split, encoded in the same way as in progressive meshes. Three pointers are then maintained to encode the forest. The mechanism proposed in [ESV99] is valid for hierarchies built through full-edge collapse (i.e., when vertex v is different from both v' and v''). An external memory data structure based

on a block partition of the view-dependent tree is described in [ESC00].

A different edge-based LOD data structure, called a *FastMesh*, has been proposed in [Paj01], which is specific for LOD meshes generated through half-edge collapse. In FastMesh, a variant of the half-edge data structure is used to encode the triangle mesh, and a forest of half-edges is stored, where each node represents a half-edge collapse operation. The forest of half-edges requires half the number of nodes compared to a binary forest of vertices.

Multiresolution techniques are a powerful tool for graphics acceleration in visualization of complex environments, but extracted meshes usually do not take full advantage of the underlying graphics hardware, which usually supports triangle strips. In [ESAV99], a technique is proposed, called a *Skip Strip*, that keeps a triangle strip structure during selective refinement.

3.3.4 Variable-resolution LOD Models for 3D Scalar Fields

In this Section, we present data structures developed for models of 3D scalar fields described by tetrahedral meshes. Such models are built from volume data sets (see also [DDFMP01b]).

3.3.4.1 Regular LOD Representations

In the finite element and computer graphics literature, there has been a burst of research on nested tetrahedral meshes generated through tetrahedron refinement. *Red/green triangulation methods* subdivide a tetrahedron σ into eight tetrahedra, four of which are obtained by cutting off the corners of σ at the edge midpoints and are congruent with the parent. The remaining four tetrahedra are obtained by splitting the octahedron resulting from cutting σ [Bey95, GLE97, Zha95]. There are different ways of splitting such octahedron which may result in a different number of congruent tetrahedral shapes. A tree of tetrahedra is used to describe the nested structure of such meshes. To reduce the number of congruent shapes generated by the subdivision process, a domain partition technique into tetrahedra and octahedra has been introduced in [GG00]. A tetrahedron σ is partitioned into four congruent tetrahedra as before and into an octahedron, which is in turn subdivided into six octahedra and eight tetrahedra. Greiner and Grosso [GG00] show that this refinement rule generates only two congruent shapes. As in the case of 2D meshes, if selective refinement is performed, irregular refinement rules for tetrahedra and octahedra must be introduced. In [GG00], nine types of edge refinement patterns are shown to be necessary for irregular tetrahedron refinement.

Another common way of generating nested meshes consists of recursively bisecting tetrahedra along their longest edge [Mau95, RL92, Pas02]. *Tetrahedron bisection* consists of

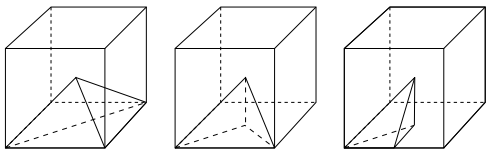


Figure 3.9: (a) Subdivision of the initial cubic domain into six tetrahedra. Examples of (b) a 1/2 pyramid, (c) a 1/4 pyramid, and (d) a 1/8 pyramid (see [LDFS01]).

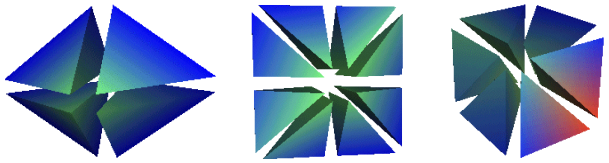


Figure 3.10: (a) Diamond formed by four 1/2 pyramids. (b) Diamond formed by eight 1/4 pyramids. (c) Diamond formed by six 1/8 pyramids (see [LDFS01]).

replacing a tetrahedron σ with the two tetrahedra obtained by splitting σ at the middle point of its longest edge and by the plane passing through such point and the opposite edge in σ [Heb94, Mau95, RL92]. This rule is applied recursively to an initial decomposition of the cubic domain obtained by splitting it into six tetrahedra, all sharing one diagonal. This gives rise to three congruent tetrahedral shapes, called *1/2 pyramids*, *1/4 pyramids* and *1/8 pyramids*, respectively (see Figure 3.9).

When we apply a tetrahedron bisection, all tetrahedra that share a common edge with the tetrahedron being split must be split at the same time to guarantee that a conforming mesh is generated. The tetrahedra which share their longest edge and that thus must be split at the same time form a *diamond* [GDL$^+$02, LDFS01, Pas02]. There are three types of diamonds generated by the three congruent tetrahedral shapes, which are formed by 1/2, 1/4 and 1/8 pyramids, respectively (see Figure 3.10).

As in the case of surfaces described by regular LOD models, the data structures for encoding nested regular meshes produced through tetrahedron bisections encode (implicitly or explicitly) either a binary forest of tetrahedra, that called a *Hierarchy of Tetrahedra (HT)*, or as a collection of diamonds with their direct dependencies.

In an HT, the roots correspond to the six tetrahedra in which the initial cube is subdivided. Any other node describes a tetrahedron σ and the two children of σ are the two

tetrahedra obtained by splitting σ along its longest edge. Each of the six trees is a full binary tree which can thus be encoded implicitly. A hierarchy of tetrahedra directly extends the triangle bintree data structures used for LOD models based on triangle meshes (see Subsection 3.3.3.1). Note that a forest of tetrahedra does not need to be explicitly encoded unless approximation errors, or other attribute information must be associated with tetrahedra.

In order to extract conforming meshes from an HT, error saturation techniques [GR99, ZCK97] have been applied, thus implicitly forcing all parents to be split before their descendants (see also [LP02] for an effective saturation technique for terrains). A very efficient alternative consists of assigning to each tetrahedron a *location code*, with a mechanism similar to that adopted for linear encoding of triangle quadtrees [LS00], or triangle bintrees [EKT01]. Location codes are not stored, but they are computed when extracting a mesh during selective refinement. In [Heb94] parents, children, and neighbors of a tetrahedron in a nested tetrahedral mesh are computed in a symbolic way, but finding neighbors still takes time proportional to the depth in the hierarchy. A worst-case constant-time neighbor finding technique has been proposed in [LDFS01]. The experimental comparisons, described in [DFL03], performed on the basis of mesh extractions at a uniform resolution, have shown that the meshes extracted using error saturation have, on average, 5% more tetrahedra than those extracted with the neighbor finding algorithm. On the other hand, the computing times required for extracting the mesh are the same for both the saturated and non-saturated versions. An analysis and experimental comparison with LOD models based on irregular meshes has been performed in [DDFLS02a] and is reported in Chapter 5.

In [GDL$^+$02], a data structure is proposed based on the direct encoding of the diamonds corresponding to the three classes of tetrahedral shapes generated by the tetrahedron bisection process. We call this data structure a *DAG of diamonds*. It extends the DAG of vertex dependencies discussed in Subsection 3.3.3.1. The root of the DAG is the initial subdivision of the cube (a non-aligned diamond), any other node is a diamond and the arcs describe the parent-child relation. Given a diamond d, the *parents* of d are those diamonds that must be split to create the tetrahedra of d. The diamonds that are created when d is split are the *children* of d. In [GDL$^+$02], a compact data structure for encoding a DAG of diamonds is described in which the DAG structure has not to be explicitly recorded. Diamond information as well as error information are attached to each vertex together with its field value. Only three Bytes per diamond are used to store the precomputed error information for the diamond.

3.3.4.2 Irregular LOD Representations

An LOD model based on an irregular tetrahedral mesh can be generated through different update operations. The most common update operation is edge collapse/vertex split. Vertex insertion/removal, which is commonly used in the case of triangle meshes, is much less common for tetrahedral meshes, because of the theoretical problems involved in removing and in inserting a vertex into a tetrahedral mesh with a non-convex domain [RO96, She98] (see Subsection 3.1.2). Tetrahedron collapse to a new vertex has also been used to simplify tetrahedral meshes [CM02].

In [CDFM$^+$04], an LOD data structure for irregular tetrahedral meshes, called a *Full-Edge Tree (FET)*, has been developed to efficiently encode an LOD model generated through full-edge collapse. The direct dependency relation among updates is encoded in an FET through a view-dependent tree [ESV99]. Let us consider a full-edge collapse, which contracts an edge $e = (v', v'')$ to a vertex v which is an internal point of edge e. In the FET, implementation [CDFM$^+$04] such point is the midpoint of edge. An internal node of the view-dependent tree corresponds to a full-edge collapse and to its inverse vertex split and contains:

- an offset vector, which is used to find the positions of vertices v' and v'' from that of v, and vice-versa;
- an offset field value, which is used to obtain the field value at v' and v'' from that of v, and vice-versa;
- a bit mask, which is used to partition the set of tetrahedra incident at v.

The bit mask contains one bit for each tetrahedron incident at v. The following rule is applied: tetrahedra marked with 0 must replace v with v'; tetrahedra marked with 1 must replace v with v''; each triangular face shared by two differently marked tetrahedra must be expanded into a tetrahedron incident at both v' and v''.

The total cost of the FET has been shown to be about 1/3 of the cost for encoding the reference mesh as an indexed data structure, and about 15% the cost for encoding the reference mesh as an indexed structure with adjacencies [CDFM$^+$04]. In Chapter 5, we will also compare the FET with the compact data structures for LOD models based on tetrahedral meshes that we have proposed.

3.4 Out-of-core Techniques

Many application domains, including 3D scanning, geometric modeling, and numerical simulation, require handling very large two- and three-dimensional meshes, consisting of

hundreds of millions of simplices. Despite the rapid improvement in hardware performance, these meshes largely overload the performance and memory capacity of state-of-the-art graphics workstations. For this reason, a wide variety of simplification methods, compression techniques and multiresolution models have been proposed. This section aims to review such methods.

3.4.1 Out-of-core Simplification and Compression

Various techniques have been presented to address the problem of huge mesh simplification. In [Lin00], Lindstrom presents an out-of-core simplification algorithm for triangle meshes, which is based on the clustering algorithm by Rossignac and Borrel [RB93]. It adopts a vertex clustering approach on a uniform grid as in [RB93], but it has a lower time and space complexity, and uses a quadric error metric to improve mesh quality.

In [LS01], Lindstrom and Silva propose improvements to the out-of-core simplification algorithm presented by Lindstrom in [Lin00], which increase the quality of the approximations and reduce memory requirements. The original algorithm has memory complexity that depends on the size of the output mesh, but no dependency on the size of the input mesh. The new algorithm removes the requirement of having enough memory to hold the simplified mesh. Two other contributions improve the quality preserving surface boundaries and optimizing the position of the representative vertex of a grid cell.

The method presented in [Lin00] has also been extended by Shaffer and Garland [SG01], and resulted in an algorithm that makes two passes over the input mesh. During the first pass, the mesh is analyzed and an adaptive space partitioning, using a BSP-tree, is performed. Using this approach, a larger number of irregular grid cells (and, thus, samples) can be allocated to the more detailed portions of the surface. However, their algorithm requires more RAM than the algorithm by Lindstrom in order to maintain a BSP-tree and additional quadric information in-core.

Note that the above methods are quite efficient, but none of them can produce a hierarchical structure for level-of-detail visualization or analysis.

Garland and Shaffer in [GS02] present a technique that combines vertex clustering and iterative edge collapse. This approach works in two steps: the first step performs an uniform vertex clustering and computes the error quadrics. The second step performs an iterative edge collapse that takes into account the error quadrics computed during the first step.

Bertocci et al. [MB00] describe a radically different approach to out-of-core simplification. Their method splits the input mesh into separate patches that are small enough to be simplified individually in-core by using a conventional simplification algorithm. Special

care has to be taken along the patch boundaries. A similar technique has been proposed by Hoppe for creating hierarchical levels of detail for height fields [Hop98a], which has been later generalized by Prince [Pri00] to arbitrary triangle meshes. While conceptually simple, the time and space overhead of partitioning the model and of later stitching the various pieces together leads to an expensive in-core simplification process, making such methods less suitable for simplifying very large meshes.

El-Sana and Chiang in [ESC00] propose an external-memory algorithm to support view-dependent simplification of triangle meshes that do not fit in main memory. Similarly to [MB00, Hop98a], they segment the model into sub-meshes that can be simplified independently and later merged in a post-processing phase. Segmentation and stitching are made simple by ensuring that edges are collapsed in order of their edge length, and guaranteeing that the boundary edges of each sub-mesh are longer than interior edges. At run-time, only the portions of the view-dependence tree that are necessary to render the given level of detail are kept in main memory.

In [CMRS03b], Cignoni et al, propose an out-of-core incremental simplificator for triangle meshes. As [Hop98a, MB00], it is based on mesh partitioning and subsequent independent simplification. The idea is to subdivide the model with an adaptive octree and simplify each leaf of such tree iteratively. Once some leaves are simplified, they can be merged (since they are smaller) and simplified further. Hoppe's approach [Hop98a] simplifies each block while freezing borders and then traverses the block hierarchy bottom-up by merging sibling cells and again simplifying. In this approach some of the borders remain unchanged until the very last simplification step. In [CMRS03b] this problem is avoided.

Isenburg et al. in [ILGS03] presented a simplification technique for triangle meshes based on a sequential processing. A processing sequence represents a mesh as an interleaved ordering of indexed triangles and vertices. This representation streams very large meshes through main memory. At each step, only a small portion of the mesh is kept in-core. Mesh access is restricted to a fixed traversal order, but full connectivity and geometry information is available for the active elements of the traversal. The simplification step is performed only on the portion of the mesh loaded in main memory.

Isenburg and Gumhold in [IG03] presented a technique for compressing huge meshes. This approach is based on a half-edge out-of-core data structure that stores a triangle mesh as a particular interleaved ordering of indexed triangles and vertices. This data structure stores adjacency information and can be used to iteratively encode the whole model with the encoding scheme proposed by Touma and Gotsman [TG98]. This results in good compression ration and fast decompression speed, even with tiny memory footprint.

3.4.2 Isocontouring and Visualization of Large Datasets

Bajaj et al. in [BPTZ99], presented a technique for partitioning and representing a large regular dataset at multiple levels of granularity. This allow efficient parallel computations. The authors also present a technique for isosurface extraction and visualization that exploit this out-of-core data structure. Zhang et al. in [ZBB01] improved the technique presented in [BPTZ99]. The model is partitioned and processed on a cluster of standard PCs that extract isosurfaces with an improved algorithm that minimize I/O operations. The visualization step is performed on a cluster of graphic workstations and composed with a new piece of hardware.

Chiang and Silva [CS99], presents a new data structure, called *binary-blocked interval tree*, for partitioning regular models and performing parallel isosurface extraction. Such data structure, compared with previous interval tree and metablock tree, reduces space requirements, but slightly increases query times.

Agarwal et al. in [AAM+98] address the problem of isocontouring large irregular 2D scalar fields. They present an I/O-optimal algorithm for this problem which stores a terrain with N vertices using $O(\frac{N}{B}b)$ blocks, where B is the size of a disk block, so that for any field value, the related isocontour can be computed using $O(log_B N + \frac{n}{B})$ I/O operations, where n denotes the size of the resulting isocontour. They also present data structures and algorithms for blocking bounded-degree planar graphs such as TINs (i.e., storing them on disk so that graph traversal algorithms can traverse the graph in an I/O-efficient manner).

In [FS01], Farias and Silva presented two out-of-core techniques for rendering large unstructured meshes. The first approach is completely disk based, it requires an external memory sorting and is quite slow. The second technique requires a preprocessing step and store some additional information, but rendering times are definitely better.

In [ACW+99], Manocha et al. presented a technique for visualization of large environments. The fundamental idea is to render objects far from the viewpoint using fast image-based techniques such as impostors, and to render all objects near the viewpoint as geometry by using multiple integrated rendering acceleration techniques, including several types of culling and levels-of-detail. The model is partitioned into viewpoint cells and associates with each cell a cull box. The cull box defines the separation into near and far geometry.

The previous idea has been improved by Varadhan and Manocha in [VM02]. In this case the model is represented through a scene graph. The idea is to load and render only the portion of the scene within the field of view. Each node of such graph is associated to different representations of the same object. They used a few meshes at different LODs but it is possible to use impostors, discrete or continuous LOD. Their implementation is based on two treads: the first one performs I/O operation, the second is in charge of the rendering step. Thanks to different LOD representation of objects, view frustum

and occlusion culling and an efficient prefetching algorithm, this technique achieves good performances comparable to in-core methods.

3.4.3 Out-of-core Multiresolution Modeling

. Lindstrom and Pascucci in [LP01, LP02] present a multiresolution technique for representing and rendering large terrain datasets stored in external memory. The hierarchical model technique is a bintree and error saturation is applied in order to maintain mesh consistency. Query and visualization times are improved by an efficient view culling and accurate error metric.

DeCoro and Pajarola in [DP02] propose a technique to store and query multi-resolution models based on half-edge collapse. They do not present an algorithm for simplifying the mesh in external memory, but they focus on the construction of the model and on its partitioning into blocks. Each block contains a possibly balanced binary tree, in which each node is associated with a vertex-split operation. Both the simplification and the construction of the multi-resolution model are to be performed in-core.

El-Sana and Bachmat in [ESB02] improved the view-dependent data structure presented in [ESV99], in order to be able to handle large multiresolution models and reduce query times. The idea is to impose a spatial subdivision over the view dependent tree, thus reducing the set of nodes to be checked during selective refinement. The rendering step is accelerated by using vertex arrays.

Lindstrom in [Lin03] proposes an out-of-core multiresolution model based on the out-of-core simplification technique described in [LS01]. Construction of the multiresolution model requires two steps. During the first step, the mesh is regularly sampled and simplified. The second step builds an octree. Leaves of the octree are associated with vertices created by the first step, and are further collapsed in lower level of the octree. Triangles are stored into the nodes introducing them.

In [CGG+04], Cignoni et al. describe a technique for out-of-core construction and view-dependent visualization of large triangle based multiresolution meshes. In order to be able to handle huge meshes the domain is decomposed according to a hierarchy of tetrahedra. Leaves of such hierarchy are associated with portions of the mesh at full resolution, while internal nodes are associated with simplified patches. This multiresolution representation is created during a fine-to-coarse simplification process. The view-dependent visualization starts from the root of the hierarchy of tetrahedra and iteratively select a subset of the refinement steps. The simplification process is based on Hoppe's method [Hop99] but it is adapted in order to keep boundary coherence among adjacent tetrahedra. Each patch is subdivided with strips and stored in secondary memory with a compact encoding [IS01].

3.5 Extraction of Morphology from Discrete Scalar Fields

In this Section, we review and analyze algorithms for extracting an approximation of the Morse-Smale complex from discrete data describing scalar fields. Most of the literature, as well as our contribution described in Chapter 7, is focused on two-dimensional scalar fields (i.e., terrains), while fewer proposals exist for morphological analysis of 3D scalar fields (see Subsection 3.5.4). An extensive survey and analysis of such techniques can be found in [CFPD04].

An approximation of the Morse-Smale complex is obtained either by fitting a C^1- or C^2-differentiable surface on a discrete dataset describing a terrain, or by simulating a Morse-Smale complex on a piecewise-linear interpolation of the terrain. A terrain dataset consists of a finite set of elevation data given at a set of points not necessarily sampled from a Morse function, thus, not necessarily with isolated critical points. Note that information on the gradient and higher order derivatives are not available. An approximation of the Morse-Smale complex in the discrete case can also be computed by applying the discrete watershed transform. The watershed transform in the C^2-differentiable case provides a decomposition of domain of function f into the unstable manifolds of the minima, that is an unstable Morse complex. By a change in the sign of the elevation function, the stable manifolds of the maxima can be obtained, and thus the stable Morse complex for the original function. The overlay of the two generates the Morse-Smale complex.

Note that, if the elevation function f is a C^2-differentiable function, then the integral lines do not intersect, and, for every point p in the graph of f, there is only one integral line that passes through p. This is not true for piecewise-linear functions. Let us consider a piecewise-linear function f defined over a triangulated domain, and two points p and q on the graph of f. Then, if the line of steepest descent from p contains q, then the line of steepest ascent from q does not necessarily contain p. This means that the lines of steepest ascent and descent may intersect, as noted also in [Sch05]. One possible intersection is illustrated in Figure 3.11, where the line of steepest descent (from point q at elevation 23 to point p at elevation 10) intersects the line of steepest ascent (from point s at elevation 17 to point r at elevation 30) at point t of elevation 20.

The stable (unstable) manifold of a point p in a Morse complex is defined as the set of points q on the graph S of f, such that the ascending (descending) integral lines from q reach p. As a consequence of this definition, the boundaries of stable (unstable) 2-cells are integral lines. These latter two conditions cannot be both guaranteed in the piecewise-linear case. If the definition of the stable (unstable) complex is *region-based*, i.e., as the set of points q such that lines of steepest ascent (descent) reach the same critical point, then it cannot be guaranteed that the boundaries of stable (unstable) 2-cells are lines of steepest

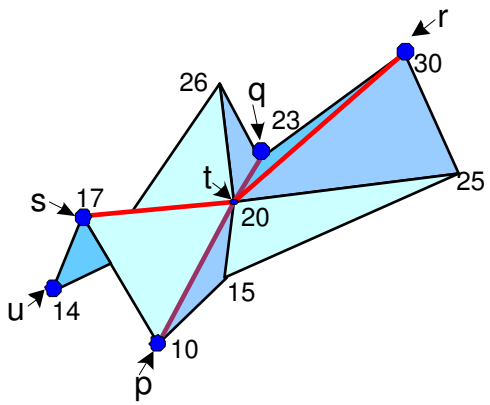


Figure 3.11: Lines of steepest ascent connects the points s, t, r and descent connects the points q, t, p. Numbers indicate elevation values.

descent (ascent) that emanate from saddle points. If, on the contrary, the definition of the Morse-Smale complex is *boundary-based*,i.e., as the set of points in regions bounded by descending (ascending) separatrix lines that emanate from saddles, then it cannot be guaranteed that a stable (unstable) 2-cell of a critical point p contains all points, and only those points, for which the line of steepest ascent (descent) ends in p. For example, in Figure 3.11, a point s at elevation 17 is a saddle point. One of the lines of steepest ascent starts from s, goes through a regular point t at elevation 20, and ends in a maximum r at elevation 30. This line separates the domain into unstable components of the two minima u and p at elevation 14 and 10, respectively if the boundary-based definition is adopted. Let us consider now a point q at elevation 23. It belongs to the unstable component of u, but the line of steepest descent from q reaches the minimum p, and not u.

Most of the algorithms proposed in the literature extract and classify critical points and then trace the integral lines starting from saddle points, until a maximum or a minimum is reached. Some of these algorithms are based on regular grids such as [BPS98, SW04, Sch05], others are based on triangle meshes as [TIKU95, EHZ01, BS98, BEHP03, Pas04]. Watershed algorithms based on simulated immersion [VS91], topographic distance [Mey94] and on rain-falling simulation [MW99] compute the regions on the terrains forming the catchment basins. Watershed algorithms by topographic distance and by rain-falling simulation adopt the definition of a catchment basin of a minimum, which is analogous to the definition of an unstable component of a Morse complex, i.e., as a set of points for which the path (integral line) of steepest descent ends in the same minimum.

3.5.1 Extracting Critical Points

In this Subsection, we present methods which can be used to extract only critical points, or label terrain data (without extracting any critical lines). Several techniques focus on extracting critical points from regular grids [PD75, TF78, WLH85], most of which have been developed initially for gray-scale images. In [Ban70, NGH04], a technique is proposed for extracting critical points from a triangulated terrain. This technique is used by most of the algorithms which compute the Morse-Smale complex based on a triangle mesh (described in Section 3.5.3).

A point p is classified as a critical point (maximum, minimum, saddle), or as a point which belongs to a critical line, based on the values of the elevation function f at p and at the points in some neighborhood of p. This means that the characterization and classification of characteristic points is done with a *local* criterion and, thus, for instance, can be done in parallel. This is similar to the C^2-differentiable case, where a critical point p is characterized and classified based on the partial derivatives of f at p, which are also local information.

On the other hand, it is not possible to characterize the points which belong to separatrix lines based only on local considerations. Those methods, which are based on a regular grid can only extract points which belong to crest and course lines, since these features can be characterized locally (for example, by considering principal normal curvature k_1 and k_2 of f at p). A line c is a *crest (course) line* if each point p on c is a local maximum (minimum) for the restriction of f to the line obtained as the intersection of the surface S associated with f and the vertical plane at p normal to the projection of the tangent vector of c at p to the xy plane. The set of ridge and valley lines is the subset of the set of crest and course lines, but the reverse is not true.

The method in [PD75] uses the relative elevation values at points in a 4- or 8-connected neighborhood [Soi04] of a point p, and considers the number of the sign changes of the (signed) difference between the elevation at p and elevation at neighboring points. The neighborhood of p is scanned in a clockwise, or counter-clockwise, order and the number of sign changes from the points at lower elevation to the points at higher elevation and vice-versa is counted, as well as the number of points in each contiguous sequence of points at lower (higher) elevation.

The method in [TF78] uses more elaborate concepts for classifying a point p, thus producing a set of uniformly labeled regions. It introduces two quantities, namely the connectivity number CN and the coefficient of curvature CC, which are local in nature, since they are computed from elevation values at p and at the points in some neighborhood of p. Intuitively, the connectivity number CN at a point p measures how many connected components, with an elevation greater than the elevation at p, exist in the neighborhood of p. The coefficient of curvature CC measures the sum of angle changes, in the neighborhood of p, of the contour line with elevation $f(p)$, measured counterclockwise. Characteristic

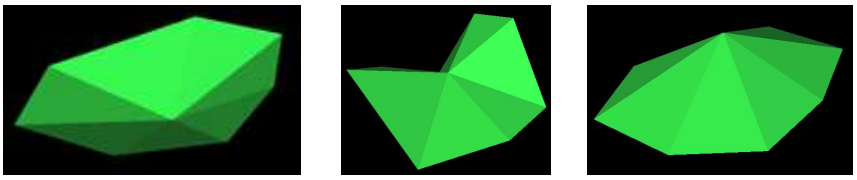


Figure 3.12: (a) a minimum. (b) a saddle point. (c) a maximum

regions are extracted as the connected components of points belonging to the same class.

The method in [WLH85] uses approximations of the initial data points through either a generalized B-spline, or a discrete cosine transformation. These approximations, which are C^0-differentiable inside the neighborhood of p and which are C^2-differentiable at p, are built based on the elevation values at p and at its neighboring points.

Since all the three methods are heuristic in nature, the quality of the output they produce (especially when they classify a point as belonging to a critical line) depends on the input data set. The first two methods [PD75, TF78] do not assume any approximation function, and also the third method [WLH85] uses a globally discontinuous approximation.

The method described in [Ban70, NGH04] work on a triangle mesh by considering the link of each vertex p of the mesh. $Lk(p)$ can be decomposed into the union of three different subsets of edges, namely $Lk^+(p)$, $Lk^-(p)$ and $Lk^\pm(p)$. $Lk^+(p)$, called the *upper link*, consists of the set of edges $[p_i, p_j] \in Lk(p)$ such that $f(p_i) > f(p), f(p_j) > f(p)$. $Lk^-(p)$, called the *lower link*, consists of the edges $[p_i, p_j] \in Lk(p)$ such that $f(p_i) < f(p), f(p_j) < f(p)$, while $Lk^\pm(p) = [p^+, p^-] : f(p^+) > f(p) > f(p^-)$ is the set of mixed edges. In this way, if $Lk^-(p) = 0$, a point p is a minimum, if $Lk^+(p) = 0$, p is a maximum, if $Lk^\pm(p) = 2$, p is a regular point, while if $Lk^\pm(p) = 2 + 2n$, p is a saddle with multiplicity n (n-fold saddle). Figure 3.12 shows a minimum (a), a saddle (b) and a maximum (c), respectively.

3.5.2 Extracting a Morse-Smale Complex from a Regular Grid

In this Subsection, we describe methods that, starting from a regular grid, extract an approximation of the Morse-Smale complex. These methods can be subdivided into two main categories:

- methods which compute the critical net on an approximating surface defined on the grid;
- methods for computing the discrete watershed transform, which are applied to compute the stable and unstable Morse complexes.

3.5.2.1 Methods for Computing the Critical Net

In this subsection, we present three boundary-based methods [BPS98, SW04, Sch05]. They first extract critical points, and then compute the separatrix lines as lines of steepest ascent (descent) starting from the saddle points. All the three methods try to fit a surface of a certain degree of continuity to the input data.

The interpolating surface used in [BPS98] is a globally C^1-differentiable Bernstein-Bezier bi-cubic function. This interpolating function does not remove any critical point of the initial input data. In order to ensure that the number of newly introduced critical points is small, a "damped" central differencing scheme for computing the coefficients of the interpolating function is devised. The basic idea of "damping" is to keep the interpolant monotone inside the grid cells whenever possible. Integral lines are computed through a Runge-Kutta technique [PTVF92]. Four separatrix lines are traced, from each saddle point, in the direction of the appropriate eigenvectors. Computation of a separatrix line ends when the line reaches a neighborhood of another critical point, or the boundary of the grid.

The method proposed in [SW04, Sch05] uses a bilinear C^0-differentiable interpolating function $f(x, y) = axy + bx + cy + d$. Coefficients a, b, c, d are obtained unambiguously for each 2-cell from the elevation values of the vertices of the 2-cell. This interpolating function cannot introduce additional minima or maxima (for $a \neq 0$), so minima and maxima can only occur at the vertices of the RSG, but it may introduce additional saddles inside cells at a point with coordinates $(-\frac{c}{a}, -\frac{b}{a})$. A grid point p is classified by considering only the elevation of its 4-adjacent neighbors, while a 2-cell, which contains a pass, can be detected by considering the elevation of its four vertices. Separatrix lines are traced and constructed point by point and they can follow grid edges, or go through 2-cells. When a separatrix line crosses a grid cell, it can be approximated with small (linear) steps, or computed exactly, by solving a linear system of differential equations. The exact solution (integral line) is a hyperbolic function inside a grid cell.

The second method described in [SW04] uses a bi-quadratic approximation $f(x, y) = ax^2 + by^2 + cxy + dx + ey + f$ to the input data. This approximating function is constructed by fitting a bi-quadratic polynomial to the 8-connected neighbors to each point of the grid. The method produces a globally discontinuous approximation, formed by local surface patches.

Unlike the method in [BPS98], the methods in [SW04, Sch05] compute the first and second derivatives analytically. The second method proposed in [SW04] uses this information to trace the separatrix lines, while the first method in [SW04] proceeds numerically in a step-by-step manner.

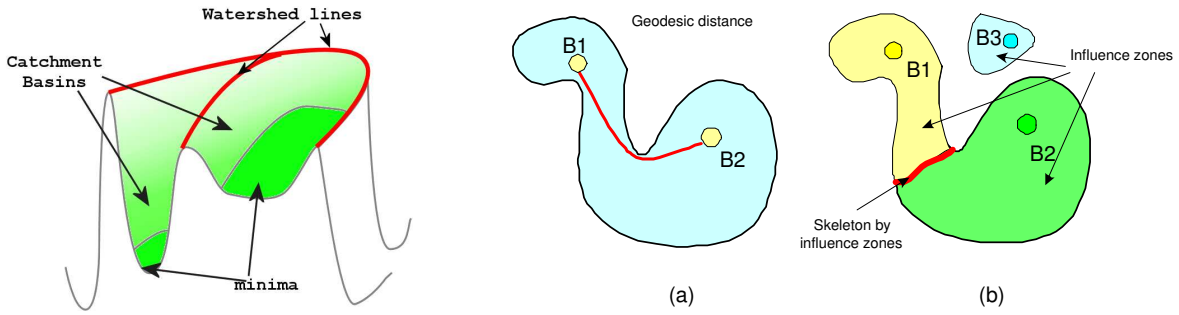


Figure 3.13: The catchment basins (green) of two minima and the related watershed lines (red) are depicted on the left. (right) (a) The geodesic distance between two points in A. (right) (b) The influence zones and the skeleton by influence zones of B with respect to A.

3.5.2.2 Watershed Methods

All watershed-based approaches start by first extracting the minima and then assigning points of the terrain to catchment basins related to the minima. Points that are not assigned to any catchment basin belong to watershed lines. A similar procedure can be applied to the same RSG and the elevation function $-f$, starting from the maxima. By computing the overlay of the two segmentations we obtain an approximation of the Morse-Smale complex. Figure 3.13 (left) shows the catchment basins of two minima and the related watershed lines.

Basically two techniques have been developed for computing the watershed transform in the discrete case starting from a regular grid: watershed methods based on *simulated immersion* [VS91], and on *discrete topographic distance* [Mey94]. The method in [Mey94] extends the idea of topographic distance from the continuous to the discrete case, while the concept of simulated immersion is defined only in the discrete case [VS91].

The idea of simulated immersion can be described in an intuitive way. Let us consider a terrain and assume to drill holes in place of local minima. We assume to insert this surface in a pool of water, building dams to prevent water coming from different minima to merge. Then, the watershed of the terrain is described by these dams, and the catchment basins of the minima are delineated by the dams. The method, that belongs to this class, uses the concept of skeleton by influence zones [Soi04] in order to define catchment basins and watershed lines.

To understand the concept of *skeleton by influence zones*, we can imagine a set $A \subseteq \mathbb{R}^n$ (or $A \subseteq \mathbb{Z}^n$) and a set $B \subseteq A$ composed of n connected components $B_1, \ldots, B_n$. The skeleton by influence zones is the set C of points in A which are equally close (in the sense of geodesic distance) to at least two connected components of B. We recall that the geodesic distance between two points p and q in A is the length of a minimal path which connects p

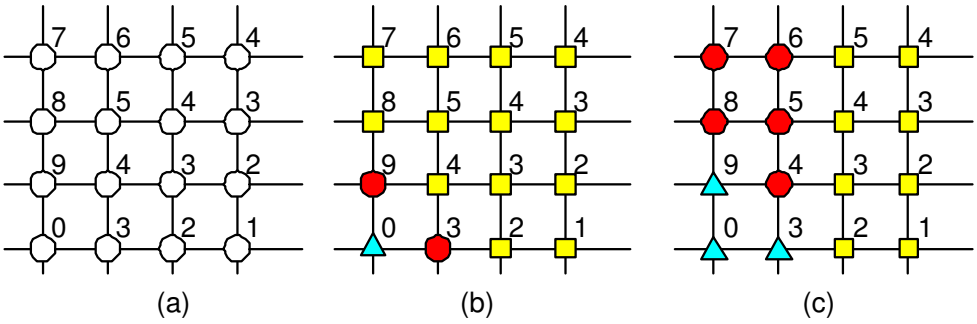


Figure 3.14: (a) The initial RSG, (b) the result applying simulated immersion, (c) the result using a watershed technique by topographic distance. Both use a 4-connected neighborhood [RM00]. Red circles belong to the watershed, while yellow squares and blue triangles identify catchment basins of the minima at elevation 1 and 0, respectively.

to q and stays within A. The *influence zone* of a component $B_i \in B$ is the set of points in A which are closer to B_i than to any other connected component B_j of B. Note that the skeleton by influence zones C of B within A is the complement of the union of influence zones of B_i within A (see Figure 3.13 on the right (a) and (b)).

The method in [VS91] recursively extracts catchment basins and watershed lines, starting from the minimal value of the elevation function f and going up. At each level of recursion, new minima can be found, or already created catchment basins are be expanded. The expansion process continues until, at a given level h, a potential catchment basin CB_h (related to level h) contains at least two catchment basins (for example CB_{ih-1}, CB_{jh-1}) already is present at level $h-1$.

This is the case in which the definition of skeleton by influence zones comes up: CB_h is partitioned into three elements, the two influence zones of CB_{ih-1} and CB_{jh-1} and the set of points in CB_h equally distant from CB_{ih-1} and CB_{jh-1} (skeleton by influence zones). The influence zones of CB_{ih-1} and CB_{jh-1} will be part of the final set of catchment basins in the output of the algorithm. The process stops when the maximal level is reached and, as usual, the watershed is defined as the complement of the set of catchment basins in the RSG.

As mentioned above, there exists another watershed technique which uses the concept of topographic distance on a regular grid. In [RM00] it is shown that the two approaches can produce different results from the same RSG as input. Figure 3.14 (a) shows the initial grid while Figure 3.14 (b) reports the result obtained by applying simulated immersion and Figure 3.14 (c) the result obtained by using watershed topographic distance. In both examples, we use a 4-connected neighborhood, for simplicity. In the following, we will describe a watershed method [Mey94] which applies the definition of topographic distance in the discrete case.

The topographic distance between two points p and q in the discrete case is defined as the minimum sum of cost among all the possible path from p to q in the grid [Mey94]. A cost is associated with each edge in a regular grid, computed on the basis of the *lower slope* $LS(p)$), defined as the maximal slope linking p to any of its neighbors of lower elevation. As in the continuous case, a path P from p to q is a path of steepest descent when the topographic distance between p and q is equal to the difference in elevation between p and q. Otherwise, the topographic distance between p and q is greater than the difference in elevation between p and q. The definition of a catchment basin in a regular grid is similar to the definition in the continuous case. The only difference is that the continuous topographic distance is replaced with the (cost-based) discrete topographic distance. The problem with this approach is that watersheds can be thick.

3.5.3 Methods for extracting Morse-Smale complex from a triangle mesh

In this Subsection, we review methods for extracting an approximation of a Morse-Smale complex from a piecewise linear approximation of a terrain defined by a triangle mesh. In Subsection 3.5.3.1, we consider methods, which extract a critical net from a triangle mesh, while in Subsection 3.5.3.2, we review a watershed method for triangle meshes.

3.5.3.1 Methods for Extracting a Critical Net

All such methods start by extracting the critical points. The methods in [TIKU95, EHZ01, BEHP03, Pas04] use the technique described in Section 3.5.1, while the method in [BS98] uses the normals of the triangles incident at point p in order to classify p. Then, all the methods extract separatrix lines. Starting from the saddle points, two lines of steepest descent and two lines of steepest ascent are constructed. The methods in [TIKU95, EHZ01, BS98] extract the separatrix lines along the edges of the TIN. The methods in [BEHP03, Pas04] compute the gradient along edges and triangles, and create lines that may cross the triangles of the TIN.

The methods in [TIKU95, EHZ01], after extracting the critical points, unfold the saddles in such a way that a k-fold saddle p is replaced with k simple saddles $p_1, \ldots, p_k$. The procedures in [EHZ01, TIKU95] differ in the way they connect the neighboring points of a multiple k-fold saddle p to k simple saddles into which p is decomposed.

The difference between the path tracing procedure of [TIKU95] and [EHZ01] is that, in [TIKU95] the neighboring point of any given point with highest elevation is chosen, while in [EHZ01] the steepest edge is chosen at every point. Note that the two approaches produce two different critical nets. Moreover, in [EHZ01], the resulting Morse-Smale complex is

further modified by using a sequence of local transformations, called *handle slides*.

To overcome the limitation of implementing handle slides, the algorithm described in [BEHP03, Pas04] extracts an approximation of a critical net from a triangle mesh by crossing the triangles so that separatrix lines of the Morse-Smale complex are not constrained to be edges of the triangle mesh, but are computed along the actual paths of steepest ascent (descent).

Finally, the method in [BS98] uses a similar procedure to [EHZ01, TIKU95] for tracing separatrix lines but it adopts a different technique to classify critical points. A point p in the mesh is classified based on the normal vectors to the triangles incident at p. The gradient at p, which is undefined in the piecewise-linear case, is assumed to take a range of values based on the normal vectors of adjacent triangles: p is considered a critical point when this range of values includes a vector $(0, 0, 1)$. A critical point p is classified as minimum, maximum or saddle point, depending on whether the gradient flow is away from, towards, or towards-and-away from p, respectively.

3.5.3.2 A Watershed Method

In [MW99], a discrete approach extending the morphological watershed algorithm by rain-falling simulation is applied. This algorithm does not assume any interpolant to be defined on the triangle mesh. In the rain-falling approach, watersheds are seen as the divide lines of the domains of attraction of rain falling over the region. Thus, a point p is assigned to a catchment basin of the minimum m_i if there is a path of steepest descent from p to m_i.

The algorithm extracts all local minima as the points which have the lowest elevation compared with all their neighboring points, and assigns a unique label to each minimum. Then, pieces of surface at the same elevation (plateaus) are identified by considering connected components of the vertices of the triangle mesh that are at the same elevation. Beginning at the lowest vertex adjacent to a plateau, a descending path P is computed on the mesh until a labeled vertex q is encountered. The path is constructed by considering, at each vertex p of the plateau, the vertex adjacent to p with minimum elevation. The vertices of P and the vertices of the pieces of surface at the same elevation are labeled with the same label as q. After the plateaus have been processed, descending paths are constructed in a similar manner, starting from each unlabeled vertex. The set of the vertices of the triangle mesh is segmented into catchment basins which correspond to minima. Catchment basins are formed by considering maximal connected components of the vertices of the mesh with the same label. Finally, a region-merging process is applied, which is based on the depth of the basins, that is the difference in elevation between the highest and the lowest vertex in each basin.

The method in [VS91] which has been originally designed for regular grid can be extended

to a mesh of arbitrary connectivity, i.e., to a mesh in which each element is a node of a graph and a node can be adjacent to an arbitrary number of other nodes. Thus, it can be applied also to a triangle mesh.

3.5.4 Extracting Critical Features from 3D Scalar Fields

Much less work has been done in extracting morphological information from a three-dimensional case. Most of the research in 3D fields aimed at enhancing the visual representation of such fields. Algorithms have been proposed to extract critical points [GP00, WSHH02] for three dimensional scalar fields. In [WSHH02], critical points are used to enhance the usability of a isosurface visualization tool by displaying an isosurface navigator containing the critical iso-values. In [EHNP03], the simulation of differentiability approach has been generalized to piecewise-linear 3D scalar fields, and an algorithm is given to compute an approximation of the Morse-Smale complex in this case.

In [FAT99], critical iso-values are used for designing the transfer function: volumes containing critical isosurfaces or structures close to critical points are emphasized. In [BPS98], critical points and lines for a 2D and a 3D scalar field are visualized in combination with contour lines and isosurfaces. More work has been done on using a compact description of the topology of the isosurfaces of a scalar field, based on the contour tree. The purpose has been to build effective interfaces to help the user in data exploration [FATT00, Pas01].

Chapter 4

The Multi-Tessellation: generalities and a 2D implementation

In this Chapter, we define the *Multi-Tessellation*, a variable-resolution continuous LOD model proposed in [DFPM97]. We give a theoretical definition of the multiresolution model and of its properties as presented in [DFMP99].

A contribution of this thesis is on in-depth analysis of the fundamental operation performed on a multiresolution model, i.e. selective refinement. Here, we define the primitives which must be efficiently implemented by any data structure encoding an MT. We analyze the description of most common approaches to selective refinement and we present a pseudocode description of such algorithms based on the previously defined primitives. This offer an abstraction layer between the selective refinement procedure, and the underlying data structure describing the multiresolution model. We also present a compact multiresolution data structure for a two-dimensional MT built though vertex-insertion operation that we have proposed in [DDFMP01a].

4.1 Background

The *Multi-Tessellation* (MT) is a variable-resolution continuous LOD model which can represent multiresolution simplicial regular meshes in arbitrary dimensions [DFPM97]. The Multi-Tessellation can also be seen as a general framework, that allows to represent, in a uniform way, every variable-resolution LOD model for simplicial meshes. In Subsection 4.1.1 we give a formal definition of such framework, and we report desirable properties (see [Pup98, Mag99] for more details).

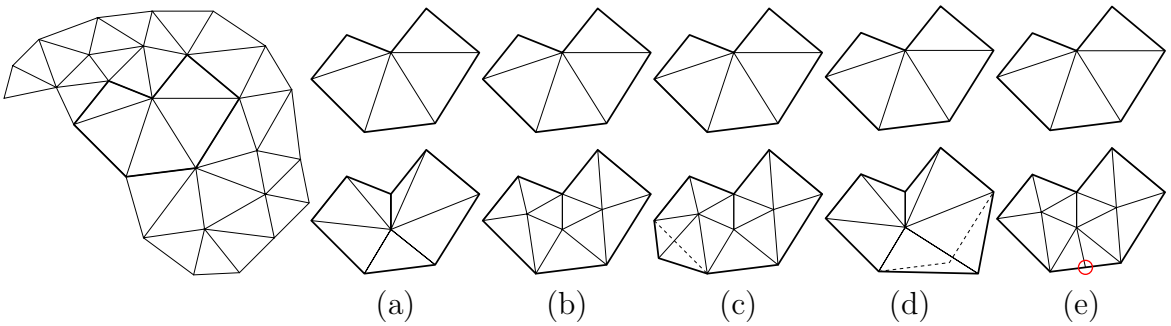


Figure 4.1: Five modifications on a simplicial complex: (a), (b) and (c) are valid while (d) and (e) are not valid since (d) violates condition 2 and (e) violates condition 3.

4.1.1 Definitions

Modifications Let Σ denote a simplicial complex, a *modification* is an operation that replaces a simplicial complex Σ_{old} (possibly contained into a larger complex Σ) with another simplicial complex Σ_{new}. The interesting case is when Σ_{old} and Σ_{new} are approximations at two different resolutions of the same (portion of a) shape. We call $u = (\Sigma_{old}, \Sigma_{new})$ a *modification.*

Applying a modification $u = (\Sigma_{old}, \Sigma_{new})$ to a simplicial complex Σ consists of replacing Σ_{old} with Σ_{new} in Σ. The result of applying u to Σ, denoted with $\Sigma[u]$, is the set of cells $(\Sigma \setminus \Sigma_{old}) \cup \Sigma_{new}$. Note that set $\Sigma[u]$ may not always be a simplicial complex. We are interested in the case where $\Sigma[u]$ is a simplicial complex. Thus, we introduce the notion of validity of a modification for a simplicial complex:

Definition 4.1.1 *Let Σ be a regular k-dimensional simplicial complex in $\mathbb{E}^d$ and $u = (\Sigma_{old}, \Sigma_{new})$ be a k-dimensional modification in $\mathbb{R}^d$. u is a valid modification for Σ if and only if:*

1. *$\Sigma_{old} \subseteq \Sigma$ (as sets of simplices);*
2. *the simplicial complex obtained by deleting Σ_{old} from Σ does not intersect Σ_{new} outside the common simplices of the boundaries of Σ_{old} and Σ_{new};*
3. *the boundary of Σ_{new} matches with the boundary of the hole created in Σ when removing Σ_{old}.*

Examples of valid and invalid modifications are shown in Figure 4.1. It has been proved that the result $\Sigma[u]$ of applying a valid k-dimensional modification u to a regular k-simplicial complex Σ is either empty, or is a regular k-dimensional simplicial complex [Mag99].

For the purpose of defining a multiresolution model, we are interested in k-dimensional *refinements* , i.e., modifications where Σ_{new} contains more simplices than Σ_{old}, and in *minimal* modifications, i.e., modifications that cannot be split into two or more valid modifications to be performed in a sequence. Modifications that induce changes in the topological type of the shape (e.g., increasing or decreasing its genus, merging or splitting connected components) are allowed.

In a refinement $u = (\Sigma_{old}, \Sigma_{new})$ we also use the symbol u^- to denote Σ_{old} and u^+ to denote Σ_{new}. The intuition behind these symbols is that u^- contains fewer simplexes than u^+.

Sequences of Modifications A sequence of modifications $\mathcal{S} = [u_0, \ldots u_m]$ is *valid* for a simplicial complex Σ if and only if every modification u_i in $\mathcal{S}$ is valid for the complex obtained as the result of successively applying to Σ all the modifications preceding u_i in the sequence.

We say that a modification u_i *directly depends* on another modification u_j (and, thus, u_j *directly blocks* u_i) if and only if the effect of applying u_i is removing some of the simplices introduced by applying u_j. In a set of modifications , two modifications depend on each other if they are in the transitive closure of the relation of direct dependency. In a similar way we can define when they block each other.

In a generic valid sequence a simplex σ may be created and removed several times and this creates cycles in the relation of dependency. Thus, we define a modification as *non-redundant* with respect to a set of modifications if it does not recreate k-simplices eliminated by some other modification in the set. It is easy to verify that, in non-redundant and valid sequences, each simplex either is in the initial complex, or there is exactly one modification specifying its "creation". Moreover, each simplex is either in the final complex, or there is exactly one modification specifying its "deletion".

Even if two modifications are independent, they cannot necessarily be applied to the same simplicial complex without interfering. We say that u_i and u_j are not in conflict when all the pairs of complexes ${u_i}^-$ and ${u_j}^-$, ${u_i}^-$ and ${u_j}^+$, ${u_i}^+$ and ${u_j}^-$, ${u_i}^+$ and ${u_j}^+$ intersect at most in a subset of their boundary cells, which are preserved in the modifications represented by u_i and u_j. This means that any of such pairs share the same subset of boundary cells, and the union of all their cells forms a complex. A set of modifications is thus *conflict-free* if and only if there are no conflicts between pairs modifications that are independent.

The Multi-Tessellation A Multi-Tessellation is defined by abstracting over the relation of direct dependency from a *valid, non-redundant*, and *conflict-free* sequence of *minimal refinements*. The non-redundancy of the sequence ensures that the direct dependency relation is a partial order. The validity and absence of conflicts ensure that, for every subset

of modifications closed with respect to the dependency relation, the result of applying its elements, sorted in any total order consistent with the partial one, is a simplicial complex.

Definition 4.1.2 *Let Σ_b be a d-dimensional simplicial complex called the base mesh. Let $\mathcal{S} = [u_0, \dots u_m]$ be a sequence of d-dimensional modifications such that:*

- for every $0 \leq i \leq m$, u_i is a refinement modification and is minimal
- $u_0 = (\emptyset, \Sigma_b)$.
- for every $1 \leq i \leq m$, ${u_i}^- \neq \emptyset$.
- sequence $\mathcal{S} = [u_0, \dots u_m]$ is a valid sequence of modifications for Σ_b.
- sequence $\mathcal{S}$ is non-redundant.
- set $\mathcal{U} = \{u_0, \dots u_m\}$ (the set of all modification in $\mathcal{S}$) is conflict-free.

Then, a *Multi-Tessellation* (MT) is a tuple $\mathcal{M} = (\mathcal{U}, \prec)$ where $\prec$ denotes the direct dependency relation. Clearly, such dependency relation can be represented with a DAG defined as follow: given two modifications $u_i = ({u_i}^-, {u_i}^+)$ and $u_j = ({u_j}^-, {u_j}^+)$ there is an arc (u_i, u_j) if and only if u_i directly depends on u_j.

Given a Multi-Tessellation $\mathcal{M}$, and a valid sequence $\mathcal{S}' = [u_0, \dots u_m]$ of the modifications $u_0, u_1, \dots u_n$, the *front simplicial complex* of $\mathcal{M}$ is:

$$\mathcal{F}(\mathcal{M}) \equiv \Sigma_b[u_1] \cdots [u_n] \equiv {u_0}^+[u_1] \cdots [u_n]$$

The front simplicial complex is a regular simplicial complex, and it is uniquely defined because it is independent of the specific sequence considered [Mag99]. Figure 4.2 shows a sequence of modifications , the corresponding Multi-Tessellation, and the front simplicial complex associated to the front in Figure 4.2.

A sub-MT $\mathcal{M}'$ of a Multi-Tessellation $\mathcal{M}$ identifies a subset of the modifications of $\mathcal{M}$ which contains the root u_0 and is closed with respect to the dependency relation (i.e., it contains the parents of any of its nodes). Since the modification in the MT represent refinement modifications, $\mathcal{F}(\mathcal{M}')$ is a simplicial complex at a lower resolution than the front simplicial complex of $\mathcal{M}$. Different sub-MTs of $\mathcal{M}$ provide simplicial complexes at different resolutions. Simplicial complexes at intermediate resolutions are front complexes of sub-MTs.

4.1.2 Proprieties of an MT

In what follows, we will discuss some properties of an MT which are relevant to the efficiency of its encoding data structures, and of the query algorithms operating on it, as discussed

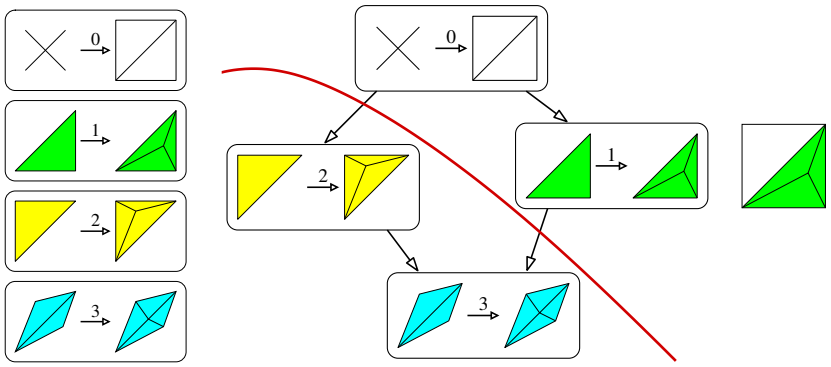


Figure 4.2: A sequence of modifications (a), an MT with direct dependencies represented with arrows (b), and the mesh corresponding to the front represented by the bold red line (c).

in [DFMP98, Mag99]. Such properties are determined by the construction algorithm used to produce the initial simplicial complex and by the sequence of modifications which form the basis of a Multi-Tessellation.

We define the *height* of a Multi-Tessellation as the maximum length of a sequence of modifications in an MT, the *width* of a modification u as the number of top simplices in u^+, and the *width* of a Multi-Tessellation as the maximum width of its modifications; the *size* of a Multi-Tessellation as the total number of top cells in it, and the *size* of the front simplicial complex of an MT as the number of its top cells.

We are interested in Multi-Tessellations which have a height logarithmic in the number of their top cells, and a width bounded from above by a small constant b. In [Pup98, Mag99], it is shown that bounded width and logarithmic height are important for the efficiency of traversal operations on an MT required in query processing, such as a point location or a range query.

We say that an MT $\mathcal{M}$ has a *linear growth* if and only if there exists a positive real number b such that for every sub-MT $\mathcal{M}'$ of $\mathcal{M}$, the ratio between the size of $\mathcal{M}'$ and the size of the front simplicial complex of $\mathcal{M}'$ is less or equal to b. It can be easily seen that if an MT has bounded width, then it has a linear growth as well.

If an MT has a linear growth, then the size of any sub-MT is linear in the size of its front complex. In particular, the size of the MT itself is linear in the size of its front simplicial complex. This means that the multiresolution structure introduces only a linear storage overhead with respect to a simple simplicial complex at the maximum resolution.

In [Pup98, Mag99], it is shown that a linear growth is fundamental to achieve optimal

time complexity (i.e., linear in the output size) for algorithms which extract a simplicial complex at a given resolution from an MT. Since these algorithms perform a traversal of the dependency relation to find the appropriate sub-MT, linear growth guarantees that the size of the traversed sub-MT is linear in the output size.

4.2 Selective Refinement Queries

Selective refinement queries consist of extracting from an LOD model a mesh satisfying some application-dependent requirements based on level of detail. The resolution of a mesh resulting from a selective refinement must obey some other application-dependent requirements, such as approximating a spatial object with a certain accuracy which can be uniform or variable in space. Here, we presents some definitions and descriptions of most common queries for 2D and 3D scalar fields.

4.2.1 Definitions

In order to formalize LOD requirements in an application-independent way, we consider a Boolean function τ, defined over the simplices of an LOD model. Function τ returns a value *true* if a simplex σ satisfies the requirements, a value *false* otherwise. We call τ an *LOD criterion*. We say also that a given mesh Σ satisfies an LOD criterion τ if and only if $\tau(\sigma) = true$, for each simplex $\sigma \in \Sigma$.

The general selective refinement query on an MT $\mathcal{M}$ can then be formulated as follows:

Given an LOD criterion τ, extract from $\mathcal{M}$ the mesh Σ_S of minimum size that satisfies τ. This is equivalent to find the front simplicial complex Σ_S satisfying τ of a sub-MT whose set of modifications has minimum cardinality (since any modification increases the mesh size).

Most common LOD criteria τ are related to an approximation error. In Subsection 4.2.2, we review some useful error metrics for scalar fields (see also [CDFM$^+$04]).

4.2.2 Error Metrics for Scalar Fields

To define an error metric, we introduce some definitions: we define a *simplicial data set* as a collection of simplices $\Sigma = \{\sigma_1, \ldots, \sigma_m\}$ having their vertices at a set of points $V = \{v_1, \ldots, v_n\}$ in an n-dimensional Euclidean space $\mathbb{E}^n$, a collection of one or more scalar field values $F = (f_1, \ldots, f_k)$, each defined at all points of V. The *size* of mesh Σ is the

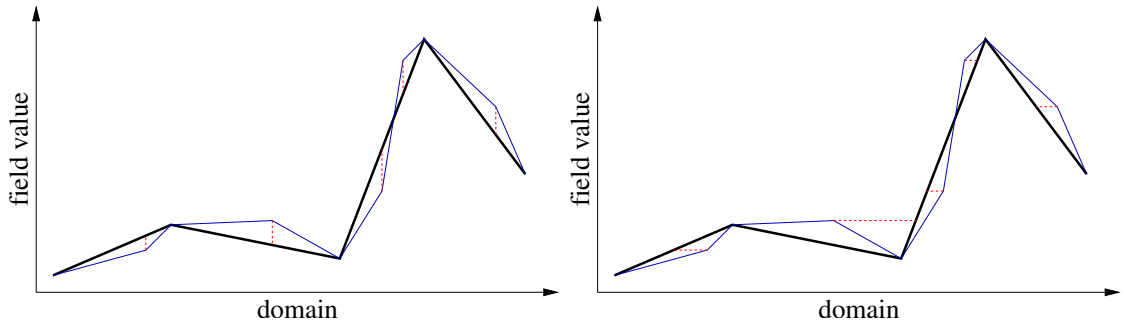


Figure 4.3: Field error (left) and iso-contour error (right) in an approximated representation of a one-dimensional scalar dataset. The original dataset is drawn with a thin line, the approximated dataset with a bold line, and the error at some relevant points is represented by a dotted red line.

number m of its simplices. For data sets used in practical applications, we can assume that $m \approx 2n$ for a 2D simplicial data set and $m \approx 6n$ for a 3D simplicial data set [CDFM+04].

We call *reference data set* a simplicial complex and a collection of field values defining the data set at the maximal resolution. An *approximated data set* is defined as a simplicial mesh Σ' with $m' < m$ simplices and with $n' < n$ vertices at a set V'. A collection of scalar fields $F' = (f'_1, \ldots, f'_k)$ is piecewise defined on Σ', similarly to F.

A common approach to measure the error associated with such an approximation is the *field error*. For every point P in the data domain, the difference between the value of the field computed at P by using the reference data set and the approximated data set is computed. This gives an estimate of how the approximated data set differs from the reference data set *(error in the input)*, but it is not always meaningful to predict how this will affect visualization *(error in the output)*. For instance, an approximated iso-contour can be arbitrarily far in space from the true iso-contour, even with a small field error, in areas in which the field has a low gradient.

For the above reason, we also use an *iso-contour* error, which is more related to error in the output. For every point P in the reference mesh, we measure its distance to the closest point P' in the approximated mesh that has the same field value as P, where the field values at P and P' are computed by using the reference and the approximated field, respectively.

Figure 4.3 illustrates the difference between field and iso-contour error metrics in a simple one-dimensional case. Such errors are computed at each cell of the approximated dataset, according to the definitions given below.

Field error We introduce first some notation. Each simplex σ' of Σ' approximates a given portion $\Omega_{\sigma'}$ of the domain of Σ, and the field F' computed on σ' approximates the corresponding field F computed on $\Omega_{\sigma'}$. We assume that F and F' span the same range of values over $\Omega_{\sigma'}$ and σ', respectively. This condition must be fulfilled while building an approximated data set. Given a point $P \in \Omega_{\sigma'}$, the point in σ' corresponding to it is the point P' that minimizes Euclidean distance from P. Thus, P' coincides with P if P is internal to σ'. Usually, P and P' are distinct only if σ' is close to the boundary of Σ. In this latter case, P' lies on the boundary of σ'.

The *field error* at σ' is defined as:

$$\varepsilon_F(\sigma') = \max_{P \in \Omega_{\sigma'}} \|F(P) - F'(P')\|.$$

Iso-contour error In order to define the iso-contour error, we define first a distance between a point $P \in \Omega_{\sigma'}$ and its closest point having the same field value in the approximated dataset:

$$d_I(P, \sigma') = \max_{f_j \in F}(\min_{P' \in \sigma' \,:\, f'_j(P') = f_j(P)} ||P - P'||).$$

Note that d_I is the largest Hausdorff distance between P and an iso-contour of value $f_j(P)$, for all j, extracted from the approximated dataset. The *iso-contour error* at σ' is defined as:

$$\varepsilon_I(\sigma') = \max_{P \in \Omega_{\sigma'}} (d_I(P, \sigma'))$$

There is a differential relation between field error and iso-contour error, which is given by the gradient of the field:

$$\varepsilon_F(\sigma') \simeq \varepsilon_I(\sigma') \nabla(\sigma')$$

where $\nabla(\sigma')$ is the magnitude of the largest gradient of a component of F' on σ'.

4.2.3 Standard Queries

This Subsection discusses some of the most common queries usually performed on 2D and 3D scalar fields (see also [CDFM$^+$04]). The first two queries can be applied to any multiresolution model whose modifications are associated with an error value. The first one selects a subset of modifications that produce an uniform resolution mesh over the whole domain. The second one produces a mesh in which the selected resolution is only in a subset of the domain. The third query is intended for scalar fields: given a field value, it extracts its iso-contours within a certain error threshold.

We define the different queries by assuming an error is associated with a modification $u = (\ u^-\ ,\ \ u^+\ )$. The (field or isocontour) error associated with u is the maximum of the errors associated with the top simplices in u^-. The queries can be defined in a completely similar way by considering the errors associated with tetrahedra.

Uniform LOD Accuracy of the extracted mesh is measured through either the field, or the iso-contour error, depending on user needs. The result of the query is a mesh having, at each simplex, an error as close as possible to a constant value E, called the *accuracy threshold*. If E is set to zero, the best approximated mesh, with respect to the error metric considered is extracted. The LOD criterion is given by:

$$\tau(u) = true \ \text{ iff } \ \varepsilon(u) \leq E$$

where $\varepsilon(u)$ denotes the maximum approximation error associated with the modification u.

Variable LOD based on a region of interest This query allows a user to focus on a certain portion of the domain. The user provides a region of interest (usually an axis-aligned box) R. Accuracy is measured either by the iso-contour error, or by the field error, depending on user needs. Accuracy should be close to a threshold E_{in} in portions of domain that intersect R, while it should be close to a coarser threshold E_{out} elsewhere. If E_{in} is set to zero, and E_{out} is set to infinity, the mesh is extracted, which best approximates the reference mesh in the focus volume. The LOD criterion is given by:

$$\tau(u) = true \ \text{ iff } \ \begin{cases} \varepsilon(u) \leq E_{in} & \text{if } \ u \cap R \neq \emptyset \\ \varepsilon(u) \leq E_{out} & \text{otherwise} \end{cases}$$

where $\varepsilon(u)$ is an approximation error associated with modification u.

Variable LOD based on the field value This query is generally useful when the user is interested in specific values, or in a range of values of the field, that we call the set of *active values* AV. Typically, the user sets an active range of field values while searching for the best value for iso-contour rendering (iso-lines in 2D or iso-surfaces in 3D) within a candidate range of values. Once the best isovalue has been identified, the corresponding iso-contour is computed on a variable LOD mesh extracted through the more appropriate isovalue-based LOD criterion. Accuracy should be close to a threshold E_{in} in cells that contain active values, while it should be close to a coarser threshold E_{out} elsewhere. If E_{in} is set to zero, and E_{out} is set to infinity, the mesh is extracted, which best approximates the reference mesh in areas close to the active values. The LOD criterion is given by:

$$\tau(u) = true \ \text{ iff } \ \begin{cases} \varepsilon(u) \leq E_{in} & \text{if } \ I_f(u) \cap AV \neq \emptyset \\ \varepsilon(u) \leq E_{out} & \text{otherwise} \end{cases}$$

where $I_f(u)$ is the range of field f spanned by simplices of u^-, and $\varepsilon(u)$ is an approximation error associated with an update u.

4.3 Algorithms for Selective Refinement

Algorithms for performing selective refinement on a multiresolution model $\mathcal{M}$ perform a traversal of the partially ordered set of modifications and construct the closed subset S of modifications, whose associated mesh Σ_S is the solution to the query. During traversal, a current closed set S of modifications is maintained, which uniquely defines a currently extracted mesh Σ_S. Set S is updated through insertion and removal of modifications according to the partial order induced by direct dependency relation, and mesh Σ_S is updated by performing the corresponding modifications. The traversal can be performed in a *top-down*, or in an *incremental* fashion [Mag99].

4.3.1 Primitives for Selective Refinement

In this Subsection, we introduce a set of primitives that are required by a selective refinement algorithm, and offer an abstraction level to the data structure implementing the multiresolution model. Every algorithm for selective requirements relies on some basic operations that perform basic steps to traverse the partial order, and on atomic operations that apply or remove a modification from a mesh, by refining or coarsening it.

While the algorithms for selective refinement are quite general, and can be adapted on most multiresolution models, implementation of such primitives is strongly dependent of the data structure adopted for the multiresolution model. Thus, we give here a description of the effect of each primitive, and, we will then describe the implementation for each data structure we will present.

1. BASE($\mathcal{M}$)
   It simply returns the first modification of the multiresolution model. The first modification in $\mathcal{U}$ is a dummy modification corresponding to creating the base mesh.

2. IS_ACCEPTABLE(u,τ)
   returns the value *True* if modification u satisfies the given LOD criterion τ, the value *False* otherwise. The LOD criterion is true when either all simplices in u^- satisfy τ (i.e. modification u satisfies τ), or that u satisfies τ, depending on whether the error values are associated with simplices or modifications. If it returns true, this means that it is not necessary to perform the refinement operation corresponding to u in order to satisfy τ.

3. PARENTS($\mathcal{M}$, S, Σ_S, u)
   given a modification u, it returns the set of modifications $u_1, \ldots u_k$ on which u directly depends.

4. CHILDREN($\mathcal{M}$, S, Σ_S, u)
   given a modification u, it returns the set of modifications $u_1, \ldots u_k$ directly depending on u. Performances of certain algorithms for selective refinement are affected by the order in which modifications $u_1, \ldots u_k$ are reported. Implementation of such primitive should carefully report modifications in the same order in which they are stored in the data structure.

5. REFINE($\mathcal{M}$, S, Σ_S, u)
   this operation adds a modification u to the current set S and performs the corresponding refinement modification on the currently extracted mesh Σ_S. In an explicit multiresolution model, (i.e., if u^- and u^+ are directly stored) it deletes from Σ_S the simplices in u^- and replaces them with those in u^+. If the modification is encoded implicitly, first it decodes the modification to retrieve u^- and u^+, and then applies it.

6. ABSTRACT($\mathcal{M}$, S, Σ_S, u)
   this operation removes a modification u from S and performs the corresponding coarsening modification on the currently extracted mesh Σ_S. In an explicit multiresolution model, (i.e., if u^- and u^+ are directly stored), it deletes from Σ_S the simplices in u^+ and replaces them with those in u^-. If the modification is encoded implicitly, first it decodes the modification to retrieve u^- and u^+ and then removes it.

Note that primitives PARENTS and CHILDREN operate on the encoding of the dependency relation, while primitives REFINE and ABSTRACT operate on the mesh extracted from the MT.

4.3.2 Top-down Algorithms for Selective Refinement

Top-down algorithms start with an empty set S and with the base mesh Σ_0, and progressively add a modification u if mesh u^- does not satisfy the LOD criterion τ. If set $S \cup \{u\}$ is not closed, then they also add all modifications which are ancestors of u and have not yet been included in S (the need for guaranteeing a closed set of updates enforces the addition of some updates that satisfy τ). A working example is shown in Figure 4.4.

The only difference between the various top-down algorithms is the policy that selects the modification u to be added. There are basically two ways according to which the top-down approach is implemented. The first one, which gives what we call a depth-first algorithm,

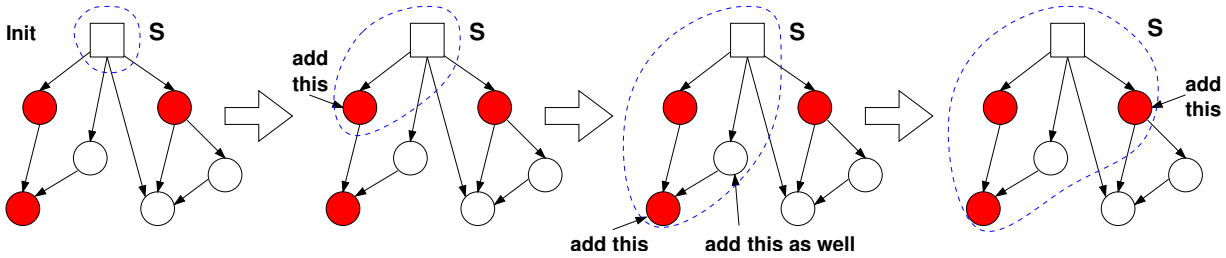


Figure 4.4: Selective refinement with a top-down depth-first approach. The square denotes the dummy modification corresponding to creating the base mesh. White updates satisfy the LOD criterion τ, gray updates do not. The dashed line encloses the current set S.

traverses the dependency relation in depth first. The second algorithm uses a best-first approach by selecting at each step the modification which is causing the largest violation of the LOD criterion.

A depth-first algorithm In the depth-first algorithm, modifications are added to the current set by following a depth-first traversal of the dependency relation. The algorithm starts from the root node ROOT($\mathcal{M}$) and refines the current mesh until a certain user-defined resolution is reached. We assume that a set and the standard primitive on sets, together with the basic primitives on the MT defined at Section 4.3.1, are available. The top-down approach to depth-first selective refinement is described by the following pseudo-code:

```
Procedure SELECTIVE_REFINEMENT(M, τ)
begin
/* create an initial empty set of modifications and empty mesh */
   set S = empty_set()
   mesh Σ_S = empty_mesh()
/* start the recursive visit from the base modification */
   TOPDOWN_DEPTHFIRST_REFINEMENT(M, τ, S, Σ_S, BASE(M))
end SELECTIVE_REFINEMENT

Procedure TOPDOWN_DEPTHFIRST_REFINEMENT(M, τ, S, Σ_S, u)
begin
/* add the modification and its missing ancestors */
   FORCE_REFINE(M, S, Σ_S, u)
   for each u' in CHILDREN(M, u) do
      if not IS_ACCEPTABLE(u', τ) then
/* start recursion on required children */
         TOPDOWN_DEPTHFIRST_REFINEMENT(M, τ, S, Σ_S, u')
end TOPDOWN_DEPTHFIRST_REFINEMENT
```

Procedure FORCE_REFINE is common to every selective refinement algorithm. It will be described here and just referred afterwords. It recursively check the transitive closure of the set of modifications applied to the base mesh, and, if required, forces the insertion of missing modifications. This can lead to insert in S also modifications that satisfy the LOD criterion specified by the user.

```
Procedure FORCE_REFINE(M,S,Σ_S,u)
/* Recursively applies all ancestors of update u that are not in S. */
begin
   P = PARENTS(M, S, Σ_S, u)
   for each u' in P do
      if u' not in S then
         FORCE_REFINE (M,S,Σ_S,u');
   endfor
   add_item(S, u)
   REFINE(M, S, Σ_S, u)
end FORCE_REFINE
```

Priority-based selective refinement Using a *priority-based* selective refinement, modifications are added to the current set S until the resolution of the mesh Σ_S becomes acceptable for the user, or the number of simplices in Σ_S reaches its maximal allowed size. Modifications are added in a priority-based fashion, in order to minimize the resolution violation, in case the maximal size of Σ_S is reached while the resolution of Σ_S is not yet sufficient. The user-requirements about resolution are expressed by primitive VIOLATION(u, τ), which returns the amount of resolution violation of simplices in u^-. A value VIOLATION(u, τ)> 0 means that mesh u^- is *under-refined* (with respect to the current requirements) and modification u should be applied. A value VIOLATION(u, τ)< 0 means the mesh u^- is *over-refined* (with respect to the current requirements) and modification u can be removed. The larger the positive [negative] value of VIOLATION, the more under-refined [over-refined] the mesh is. Primitive IS_ACCEPTABLE(u,τ) is equivalent in this case to a check: VIOLATION(u, τ)< 0.

The top-down approach to priority-based selective refinement is implemented in the following pseudo-code. Our implementation uses a priority queue, where the elements with higher value have a higher priority. Thus, the priority of a modification is its violation, since we want to give higher priority to modifications causing a larger violation of the LOD criterion. Primitive *insert_queue* adds a modification to a priority queue only if such modification is not already into the queue, otherwise the queue is unchanged. Primitive *delete_max(Q)* deletes the element with maximum priority from queue Q and returns it.

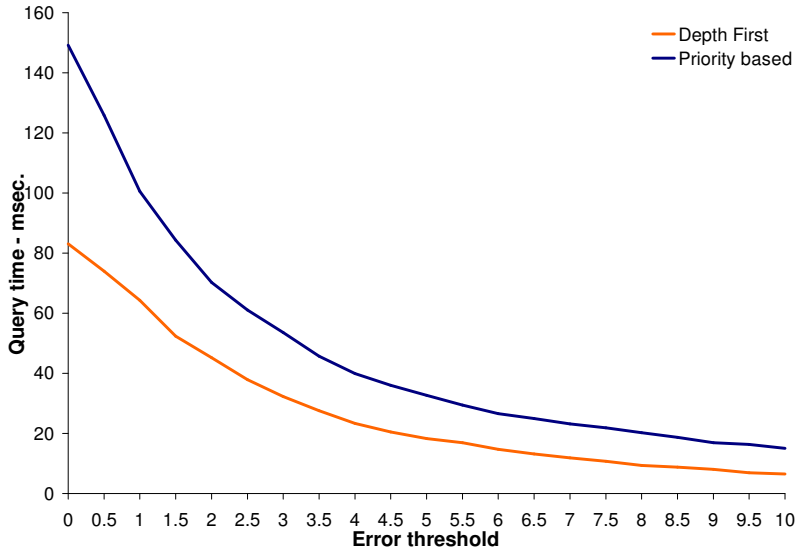


Figure 4.5: Query times with (blue line) and without (orange line) priority queue. Queries are performed at uniform resolution, with a decreasing error threshold.

```
Procedure TOPDOWN_PRIORITY_REFINEMENT(M, τ)
begin
   set S = empty_set()
   mesh Σ_S = empty_mesh()
   queue Q = empty_queue()
/* add the base modification to the priority queue */
   insert_queue(Q, BASE(M), +∞)
   while not is_empty(Q) do
/* extract from the priority queue the modification */
/* with highest violation of the LOD criterion */
      u = delete_max(Q)
/* add the modification and its missing ancestors */
      FORCE_REFINE(M, S, Σ_S, u)
      for each u' ∈ CHILDREN(M, u) do
         if not IS_ACCEPTABLE(u', τ) then
/* add required children to the priority queue */
            insert_queue(Q, u', VIOLATION(u', τ))
      endfor
   endwhile
end TOPDOWN_PRIORITY_REFINEMENT
```

We have compared the performances of the two top-down algorithms on a 2D Multi-

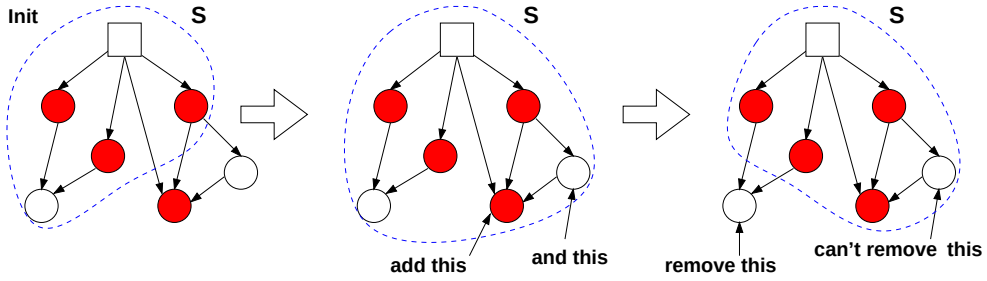


Figure 4.6: Selective refinement with an incremental approach. The square denotes the dummy update corresponding to the base mesh. White updates satisfy the LOD criterion τ, colored updates do not. The dashed line encloses the current set S.

Tessellation encoded in an explicit data structure (see Section 4.4). We have used Mount Marcy terrain data set. We have compared extraction times of the two algorithms varying the error threshold for queries at uniform resolution.

Figure 4.5 shows query time with (blue line) and without (orange line) priority queue, on the Mount Marcy dataset which is composed by 35,162 vertices and 69,718 triangles. Priority-based algorithm is, on average, 85% slower that depth first one. This overhead is partially due to the time spent in updating the priority queue. Another reason for this better performance is that the depth-first algorithm inserts a large number of modifications though procedure FORCE_REFINE, that is without checking if the LOD criterion is satisfied. This avoids costly acceptance tests, and results in more efficient query times.

4.3.3 Incremental Algorithms

When the solutions to two consecutive queries differ only slightly, an incremental approach, based on a step-by-step update of a previously computed solution, seems to be particularly suitable. The basic idea is to search for the solution to the current query by starting from the closed set of modifications generated by a previous query. Such set is updated by adding those updates u, such that u^- does not satisfy τ, and removing those, such that u^- satisfies τ, under the usual constraint that the set must remain closed. A working example is shown in Figure 4.6.

The incremental algorithm starts with a previously computed set S and mesh Σ_S, and progressively adds a modification u if u^- does not satisfy the LOD criterion τ, and than removes those, such that u^+ satisfies τ. Note that while a modification is forced during refinement, a modification is never forced while coarsening. The incremental algorithm also needs to keep track of the set F of modifications belonging to S and having at least one child not in S. In order to maintain set F we add, in the auxiliary data structure, for each modification u, the number of children of u not in S. When this number is different

from zero, it means that that modification belong to the front F.

We give below a pseudo-code description of the incremental selective refinement algorithm. The refinement part of the algorithm INCREMENTAL_SELECTIVE_REFINEMENT is the same of the depth-first algorithm. A variant of this approach using two priority queues to guide refinement and coarsening has been proposed in [DWS+97, CDFM+04].

```
Procedure INCREMENTAL_SELECTIVE_REFINEMENT(M, τ, S, Σ_S, F)
begin
/* start from a set S of modifications */
/* F contains modifications of S having at least a child not in S */
/* refinement step */
   for each u in F do
      if not IS_ACCEPTABLE(u, τ) then
/* add required modifications of F */
         INCREMENTAL_DEPTHFIRST_REFINEMENT(M, τ, S, Σ_S, F, u)
   endfor
/* coarsening step */
   for each u in F do
      C = CHILDREN(M, S, Σ_S, u)
      if IS_ACCEPTABLE(u, τ) and C ∩ S = ∅ then
/* remove modifications not required */
         ABSTRACT(M, S, Σ_S, u)
         del_item(S, u)
         del_item(F, u)
/* Add to F modifications with children outside S */
         for each u' in PARENTS(M, u) and u' in S do
            u'.num_of_child = u'.num_of_child + 1
            add_item(F, u')
         endfor
   endfor
end INCREMENTAL_SELECTIVE_REFINEMENT


Procedure INCREMENTAL_DEPTHFIRST_REFINEMENT(M, τ, S, Σ_S, F, u)
begin
/* add the modification and its missing ancestors */
   INCREMENTAL_FORCE_REFINE(M, S, Σ_S, F, u)
   for each u' in CHILDREN(M, u) do
      if not IS_ACCEPTABLE(u', τ) then
/* start recursion on required children */
         INCREMENTAL_DEPTHFIRST_REFINEMENT(M, τ, S, Σ_S, F, u')
end INCREMENTAL_DEPTHFIRST_REFINEMENT
```

```
Procedure INCREMENTAL_FORCE_REFINE(M, S, Σ_S, F, u)
/* Recursively applies all ancestors of update u that are not in S. */
begin
   P = PARENTS(M, S, Σ_S, u)
   for each u' in P do
      INCREMENTAL_FORCE_REFINE (M,S,Σ_S, F, u');
   endfor
   add_item(S, u)
   REFINE(M, S, Σ_S, u)
/* Store the number of children of u not in S */
   u'.num_of_child = | CHILDREN(M, u)|
/* Keep in F only modifications with children in S */
   for each u' ∈ PARENTS(M, u) AND u' ∈ F
      u'.num_of_child = u'.num_of_child - 1
      if u'.num_of_child = 0 then
         del_item(F, u')
   endfor
end FORCE_REFINE
```

Incremental selective refinement algorithms extract a mesh of *minimum* size, provided that there exists no modification u in $\mathcal{M}$ such that u^- satisfies τ and u^+ does not satisfy τ. This corresponds to the intuition that refining locally a mesh that is "accurate enough" cannot produce a mesh that is no longer accurate enough. The top-down algorithms always guarantee minimality, while the incremental algorithm may stop on a mesh that has a non-minimum size [Mag99].

4.4 An Explicit Data Structure for a Multi-Tessellation

In [Mag99], a dimension-independent data structure for the Multi-Tessellation has been proposed which explicitly describes the top simplices added and deleted by the modifications in a MT $\mathcal{M}$ as well as the dependency relation. We call such data structure an *explicit Multi-Tessellation* (MT).

The dependency relation is encoded as a DAG, in which the nodes are the modifications, and there is an arc (u_1, u_2) for each pair of nodes such that u_2 directly depends on u_1 (see Figure 4.7). We consider a d-dimensional MT.

4.4.1 An encoding for the dependency relation

Nodes of the MT are stored in an array, when for each node u we encode:

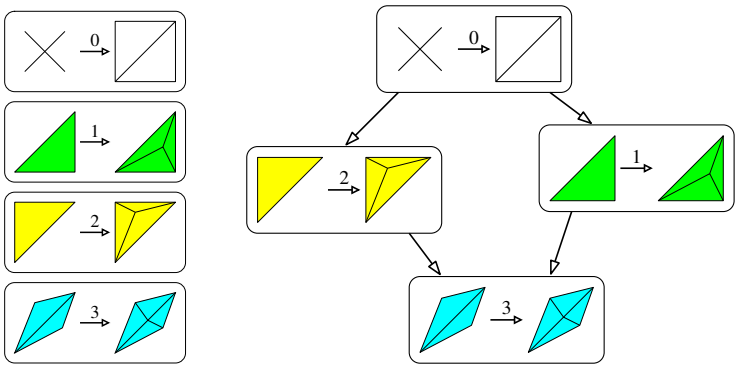


Figure 4.7: A sequence of modifications refining a triangle mesh, and the partial order depicted as a DAG: each node represents a modification, and each arc represents a dependency.

- the number of parents of u (i.e., nodes u_p such that $u_p \prec u$) and their indexes;
- the number of children of u (i.e., nodes u_c such that $u \prec u_c$) and their indexes.

We evaluate the storage cost for the case of 2D and 3D MT describing a scalar field. For each modification we store two counters and for each arc we store two indexes. The resulting cost is $2c + 8a$ Bytes, where c is the number of modifications and a the number of arcs in an MT.

A mesh Σ extracted by a selective refinement algorithm working on an MT is encoded as an indexed data structure, i.e., by storing, for each d-simplex, just the pointers to its vertices.

Primitive REFINE and ABSTRACT have, thus a straightforward implementation on such data structure. Appliyng a refinement modification $u = (u^-, u^+)$ simply consist of removing the d-simplices in u^- from the current mesh and adding those of u^+. Primitive ABSTRACT is implemented in a completely similar way.

4.4.2 An encoding for the modifications

Vertices and d-simplices of the MT are encoded in two arrays. In the array of simplices, simplices created in the same update uare stored at consecutive entries. To encode a modification, the following information need to be stored:

- geometry and field information:
  for each vertex v, its coordinates, and, if present, its field value are stored;

- connectivity information:
  for each d-simplex σ, we store the indexes of vertices of σ in the simplices array, i.e. for d-simplex σ, the indexes of $d+1$ vertices of σ are stored in an array;

- structure of the modification:
  for each modification $u = (u^-, u^+)$, we store the number of d-simplices in u^-, and the indexes of the d-simplices in u^- inside the array of simplices;
  the number of simplices in u^+, and the index of the first simplex in u^+ within the array of simplices. The d-simplices in u^+ are not explicitly stored since simplices in u^+ are stored consecutively in the simples way.

4.4.3 Storage costs of the data structure

We assume that coordinates, field values, and indexes are stored in 4 Bytes, and counters associated with a modification are stored in one Byte. Storing the geometry and field values of a model embedded in $\mathbb{E}^2$ ($\mathbb{E}^3$) requires $12n$ ($16n$) Bytes, with n number of vertices.. Storing the connectivity of a 2D (3D) model requires $12m$ for a 2D MT ($16m$ for a 3D MT) Bytes, with m total number of simplices. Encoding the structure of each modifications requires 2 counters an index for each simplex in u^- and an index to the first simplex of u^+. This results in $2c + 4c + 4m$ Bytes, where c is the number of modifications in the MT, and m is the total number of simplices in the MT. The encoding of modifications requires $12n + 6c + 16m$ ($16n + 6c + 20m$) Bytes.

We have evaluated the space complexity in two specific interesting areas, namely a 2D MT built through vertex insertion, that we call a *vertex-based 2D MT*, and a 3D MT built through half-edge collapse, that we call a *Half-Edge 3D MT*.

In a vertex 2D MT, the number c of nodes in the DAG is equal to the number of vertices. Experimentally, we found that the number m of triangles in an MT is 4 times the number of its vertices $m = 4n$, and the number of arcs at the DAG is $a = 3n$. Thus, the encoding of modifications requires $82n$ Bytes, and the encoding of the DAG requires $26n$ Bytes. The total cost of the explicit data structure for 2D vertex-based MT is equal to $82n+26n = 108n$ Bytes.

In a half-edge 3D MT, the number of nodes in the DAG is equal to n. Experimentally, for half-edge collapse, we found that the number of tetrahedra in MT is 13 times the number of its vertices $m = 13n$, and the number of arcs at the DAG is $a = 5n$. Thus, the encoding of modifications requires $282n$ Bytes, and the encoding of the DAG requires $42n$ Bytes. The total cost of the explicit data structure for 3D half-edge MT is equal to $282n + 42n = 324n$ Bytes.

A full resolution triangle mesh represented with the indexed data structure defined in

Subsection 2.2.2 requires the same amount of space for vertex coordinates and scalar field, and three indexes for each triangle. The number of triangles $m = 2n$. The total cost is $12n + 3 * 4 * 2n = 36n$ Bytes. If we store also adjacencies among triangles and a triangle incident in each vertex the total cost increases to $12n + 24n + 24n + 4n = 64n$ Bytes. In case of tetrahedral meshes (see Subsection 2.2.3) the number of indexes becomes 4 for each tetrahedron, the number m of tetrahedra is on average $m = 6n$ and the total cost increases to $16n + 4 * 4 * 6n = 112n$ with indexed data structure without adjacencies and $16n + 96n + 96n + 4n = 212n$ Bytes with adjacencies and a tetrahedron incident in each vertex.

4.5 A Compact Data Structure for a 2D Multi-Tessellation

The explicit MT has the problem of having a large overhead with respect to storing the representation at full resolution. Thus, as mentioned in the introduction, we have developed compact representations for specific instances of an MT which depend on the simplification strategy on which they are constructed. Since our focus is on multiresolution representation of scalar fields, we have developed compact representation for 2D MT generated through vertex insertion/removal and for 3D MT generated through vertex insertion/removal and through half-edge collapse.

Chapter 5 is devoted to multiresolution representation for 3D MTs, while in this section, we focus on a compact representation for a 2D MT generated through vertex insertion/removal. We call the resulting representation a vertex 2D MT. As pointed out in Section 3.3.3.2, the most common way of generating a multiresolution model for a terrain (2D scalar field) consists of starting from a coarse mesh and incrementally refining it through vertex insertion. Thus, we need a compact data structure in which the modifications are vertex insertion/removal, and which supports primitives REFINE, ABSTRACT, PARENTS and CHILDREN efficiently.

4.5.1 An encoding for the dependency relation

Klein and Gumhold [KG98] propose an interesting data structure for encoding the DAG with a reduced memory cost. However, this is paid in term of the time complexity in executing primitive CHILDREN, which is quadratic instead of linear in the output size.

Given a modification u, Klein e Gumhold consider all the modifications $\{u_1, \ldots, u_k\}$ on which u depends. They encode such modifications in a cyclic linked list, called a *loop*. The loop has $k+1$ elements: the k parents of u, and u itself. Figure 4.8 shows a modification u, the set of modifications $\{u_1, u_2, u_3\}$ from which u depends and the loop connecting them.

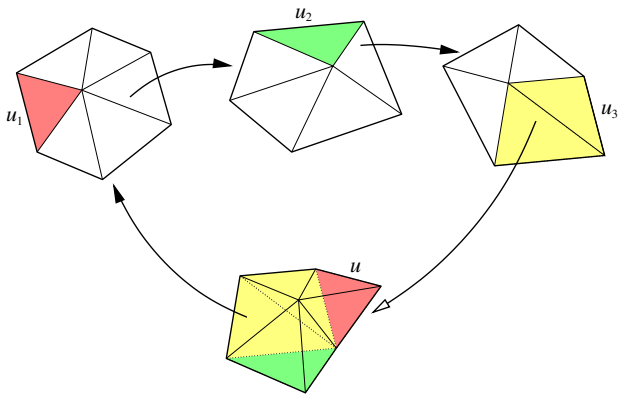


Figure 4.8: A DAG (on the left) and the loops needed to encode it (on the right).

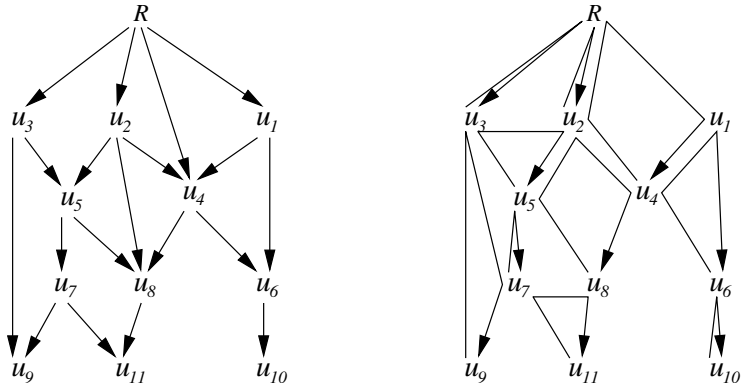


Figure 4.9: A DAG (on the left) and the loops needed to encode it (on the right).

Thus, a modification u with j children belongs to $j+1$ loops: the first one starts at u and includes the parents; the remaining j loops are those starting at the children of u. Thus, node u has $j+1$ pointers for the $j+1$ loops traversing it.

Figure 4.9 shows a DAG (left) and the loops needed to encode it (right). In this example the number of arcs is 24, while the number of modifications is 12. An explicit encoding of the DAG requires 48 pointers, while the encoding based on loops requires only 36 pointers. In order to distinguish the loop each pointer belongs to, each pointer has an index attached to it, which represents the loop identifier in the next modification. For each modification u, they also store the number of loops in which the node appears. If a node u has k parents, the corresponding loop has $k+1$ elements, each consisting of a pointer and an index.

The total number of links to describe a 2D MT is thus $a+c$. In the case of a vertex based on vertex-insertion, we can safely bound the number of direct ancestors or descendants to 15. So we need four bits for indexes and four bits to count the loops to which a node belongs. A pointer to a node requires $\log_2 c$ bits. The whole DAG encoding requires

$4c + (4 + \log_2 c) \cdot (a + c) = 8c + 4a + (a + c) \cdot \log_2 c$ bits. Experimentally, we have found that, for a vertex 2D MTs, a is about $3n$ on average and c can be approximated with n. Thus, in this case the encoding cost reduces to $20n + 4n \cdot \log_2 n$ bits.

This DAG encoding supports operations PARENTS and CHILDREN as follows. For a modification u, the set of parents is obtained by scanning the first loop. Primitive ANCESTORS returns the set of parents in linear time in the size of the set. The result of CHILDREN(u) is obtained by scanning the others j loops passing through u: the last element of each loop is a child of u. The time complexity is $(\sum(\#loop_i)) \simeq O(j^2)$, which is not optimal. Since we are using this DAG encoding also for a data structure for a 3D Multi Tessellation we report such a pseudo-code definition of such primitives in Subsection 5.1.1

4.5.2 An encoding for the modifications

When u is a vertex insertion, primitive REFINE($\mathcal{M}$, S, Σ_S, u) reduces to locating a polygon π_u on the current mesh Σ_S, which bounds the set of triangles of u^-: all triangles internal to such polygon must be deleted. The triangles of u^+ are created by simply joining each vertex of π_u to the vertex v_u inserted by u. Primitive ABSTRACT($\mathcal{M}$, S, Σ_S, u) consists of deleting all triangles incident in v_u from Σ_S (such triangles form u^+), and re-triangulating the resulting polygonal hole without using any internal point. In general, polygon π_u admits more than one triangulation, but we must recreate the one corresponding to u^-.

Thus, a family of encoding structures for modifications based on vertex-insertion/removal can be developed that do not store all the triangles in u^- and u^+ (as in the explicit data structure), but just those information that are sufficient to perform the following two tasks:

1. given the current mesh Σ_S and a modification u, such that $u^- \subset \Sigma_S$, recognize the triangles of u^- among those of Σ_S;
2. given the polygon π_u bounding u^-, generate the triangles of u^-.

In [DDFMP01a] we have studied and analyzed three different encoding techniques, and compared them to the approach proposed by Klein and Gumhold in [KG98]. One of such techniques, based on the optimal encoding of a triangulated polygon [ADF01], offers the best compression ratios but the encoding a modification is fairly complicate and the required algorithm is slow compared to the other techniques. The other two offer quite similar results in terms of compactness, but one of them is easier to be implemented and on-the-fly decompression is faster. Here, we report only the technique that presents better size/speed ratios.

Encoding the n vertices of the MT requires three coordinates per vertex, each coordinate is represented on four Bytes, with a total cost equal to $96n$ bits. Vertex coordinates can be discretized and compressed but this results in a lossy compression. Since each modification u corresponds to the insertion of a vertex v_u, modifications and vertices are re-numbered in such a way that a node u and its corresponding vertex v_u have the same label. Thus, the relation between u^+ and v_u is encoded at a null cost.

The encoding of u^- requires storing one edge of polygon π_u, which is used as the starting point for performing tasks 1 and 2 mentioned above. Such edge, that we denote as e_u, is encoded by storing the references to its two vertices, with a cost equal to $2c \cdot \log_2 n$ bits, where $c \approx n$ is the number of modifications. The remaining information, i.e., the encoding of the triangulation u^-, is described in the next subsection.

In order to perform operations REFINE and ABSTRACT, the current mesh Σ_S needs to be encoded into an indexed structure with adjacencies. In addition, in order to locate the initial edge e_u on the mesh, starting from its vertices, we need to encode a pointer from each vertex to one of its incident triangles in the current mesh. Under this assumption, the time for locating edge $e_u = (v_1, v_2)$ is linear in the degree of vertex v_1 in Σ_S (which is equal to 6 on average).

For encoding u^-, we use a variant of a technique proposed by Taubin et al. [TGHL98] in the context of progressive transmission of large triangle meshes, which describes the triangulation of a polygon by using a code of two bits for each triangle.

The construction of a bitstream B encoding u^- is performed during the construction of the model through a recursive procedure, that encodes one triangle $\sigma \in u^-$ at a time. At each recursive call, the algorithm maintains a current triangulation T_B and a current edge e_B on its boundary, and generates the code for a triangle σ bounded by e_B. Note that T_B must be represented by its triangles and their mutual adjacencies along the edges. At the beginning, the current triangulation T_B is initialized as u^- and the current edge e_B as e_u, which is the stored edge of polygon π_u. Triangle σ is the triangle of u^- lying on the left side of e_B. The algorithm looks at the two edges of triangle σ different from e_B and at the two parts in which such edges partition the current triangulation, in order to assign a code to σ and to decide about recursive calls to be made:

- if both parts are non-empty, then code 00 is added to the bitstream B for σ;
- if only the left part is non-empty, then code 01 to B is added;
- if only the right part is non-empty, then code 10 is added to B;
- if both parts are empty, then code 11 is added to B.

Then, a recursive call is performed on each of the non-empty parts, starting from the

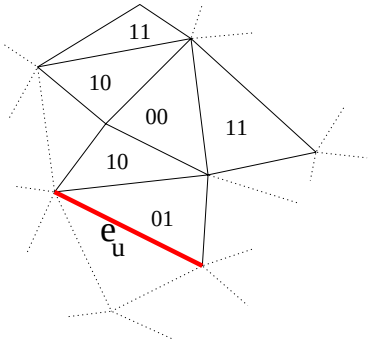


Figure 4.10: Codes assigned by Taubin's method to the triangles of a mesh u^-. The thick edge is the starting triangle $e = e_u$.

boundary edge it shared with triangle σ. On each branch, the recursion stops when a code 11 is generated. Figure 4.10 shows the codes generated for a sample mesh u^-.

Encoding a modification u, such that u^- contains h triangles, requires $2h$ bits. Note that the value of h does not need to be stored. The structure of the bitstream (basically, a preorder visit of the recursion tree, where leaves are characterized by code 11) can be exploited to reconstruct its length. Experimentally we found that on average $h = 4n$, thus, the total cost of encoding all modifications is equal to $8n$ bits on average.

In order to perform task 1 in operation REFINE, we run the same recursive procedure described above, starting from edge e_u on the current mesh Σ_S. In this case, bitstream B is known, and we need to recover u^-. At each call, we consider the current edge e_B, and recognize the triangle σ, adjacent to e, and belonging to u^-. Then, depending on the code found in B, we decide, for each edge of σ, different from e, if a recursive call must be invoked on that edge. The recursive calls progressively "consume" stream B, so that each call always reads the next code in B which has not been read by any call executed before it (either at a deeper or at an outer recursion level).

Task 2 in operation ABSTRACT is performed in a very similar way. In this case, at each step, we generate a triangle σ, instead of recognizing it on the current mesh. Note that this procedure allows reconstructing just the *topology* of triangle mesh u^-. When we create a triangle σ, we do not know which vertex of π_u must be become the third vertex of σ. However, when the topology of u^- has been built, it is easy to map the vertices of π_u onto it. This simply reduces to a synchronous traversal of π_u and of the boundary of the triangulation, starting at one of the two vertices of e_u (for which the mapping is known). For each node u, the computational cost of performing both tasks 1 and 2, with the procedures described above, is linear in the number of triangles of u^-.

		100K	1M	10M	100M.
Standard	$16 + 6 \log_2 n$	118	136	160	178
Compact	$20 + 4 \log_2 n$	88	100	116	128
Standard	$16 + 6 * 32$	208	208	208	208
Compact	$20 + 4 * 32$	148	148	148	148

Table 4.1: Costs of the DAG encoding expressed in number of bits per vertex. n denotes the number of vertices. The first two rows consider variable-length pointers, while the other two consider pointers on 4 Bytes.

		100K	1M	10M	100M
Explicit	$16 + 17 \log_2 n$	305	356	424	475
Compact	$8 + 2 \log_2 n$	42	48	56	62
Explicit	$16 + 17 * 32$	560	560	560	560
Compact	$8 + 2 * 32$	72	72	72	72

Table 4.2: Costs of the update encoding expressed in number of bits per vertex. n denotes the number of vertices.

4.5.3 Storage costs of the data structure

In this section we analyze the results achieved by adopting compact encoding for the DAG, for the modifications, or both. We compare such results with the explicit data structure for the MT and with the indexed data structure with and without adjacencies encoding the reference mesh. We also consider the encoding of a pointer with fixed length on 4 Bytes. This result in a waste of space, especially in small models, but offer faster access to linked information.

In Table 4.1, the storage costs of the standard DAG encoding and the DAG encoding proposed by Klein and Gumhold are compared by considering MTs with 100K, 1M, 10M and 100M vertices. First two rows consider pointers with variable length, while the latter two are for standard pointers encoded on 4 Bytes. The costs are expressed in number of bits per vertex. The method by Klein and Gumhold produces on average a saving, between 28% and 29% with respect to the standard DAG encoding.

In Table 4.2, the costs of encoding the modifications are compared for MTs with 100K, 1M, 10M and 100M vertices. These costs do not include vertex coordinates and the scalar field. The costs are expressed in number of bits per vertex. Compression with respect to an explicit representation of MT is about 87%. The overhead of full length pointer, for large models (n >10M) is less than 20% with respect to the compact encoding.

		100K	1M	10M	100M
Mesh	192	192	192	192	192
Mesh+Adj.	384	384	384	384	384
Explicit	$32 + 23 \log_2 n$	423	492	584	653
Explicit	$32 + 23 * 32$	768	768	768	768
Compact	$28 + 6 \log_2 n$	130	148	172	190
Compact	$28 + 6 * 32$	220	220	220	220

Table 4.3: Comparison of the overall encoding costs expressed in number of bits per vertex. n denotes the number of vertices.

In Table 4.3 the cost of the reference mesh encoded in an indexed data structure with or without adjacencies is compared with the explicit MT and to the compact one, both with short and long pointers. The explicit MT exhibit an overhead with respect to the full resolution mesh of a factor 2.8 or 4 according to the pointer adopted, while compact MT results in a moderate compression ratio (17%) if encoded with short pointer. The most interesting result is the comparison between the compact MT and the explicit MT, in this case the compression ratio is about 79%.

A vertex-based 2D MT extract up to 80K triangles per second on a Pentium III PC. Extracted mesh is encoded in an indexed mesh with adjacencies. A 2D instance of the explicit MT (see Section 4.4) can extract on the same architecture up to 380K triangles per second without adjacencies, and about 115K triangles per second with adjacencies.

Chapter 5

Compact representations for multi-resolution tetrahedral models

In this Chapter, we define, analyze and compare new compact data structures for encoding a 3D Multi-Tessellation built through specific simplification operators. With respect to the explicit MT data structure described in Section 4.4, we loose the generality, but we achieve very good compression ratios with respect not only to the explicit MT data structure, but also with respect to encoding the mesh at full resolution. For each data structure, we provide a description, an evaluation of its storage cost, and a pseudo code description of the selective refinement primitives (see Section 4.3.1).

In Section 5.1, we describe a data structure for a 3D MT built through vertex insertion or removal, that we call a *Vertex-based 3D MT* [DDFMP01b]. In Section 5.2, we describe a specialization of the vertex-based 3D MT to the case of a 3D MT built through half-edge collapse (which is a special case of vertex removal) [DDF02], that we call a *Half-Edge MT*. In Section 5.3, we describe a different data structure for 3D MT built through half-edge collapse, which is based on an extension of the view dependent tree [ESV99], that we call a *Half-Edge Tree* (HET) [SDDFM03, DFM$^+$05].

For comparison purposes, in Section 5.4, we briefly review two data structures for multiresolution models based on tetrahedral meshes, namely the *Full-Edge* MT [CDFM$^+$04] for irregular tetrahedral meshes generated through full edge collapse, and the *Hierarchy of Tetrahedra* (HT) [LDFS01] for regular meshes generated through tetrahedron bisection. We compare such data structure with the HET. A comparison between the Half-Edge MT and a Hierarchy of Tetrahedra has been presented in [DDFLS02b], while a comparison between the HET and the Full-Edge MT has been performed in [DFM$^+$05].

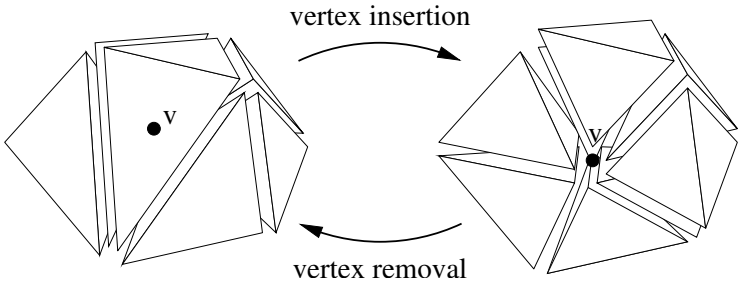


Figure 5.1: Vertex insertion and removal on a tetrahedral mesh (exploded view). On the left, the set Σ' of the removed tetrahedra is shown. On the right, the new vertex v and the new tetrahedra incident in v are shown.

5.1 A Vertex-Based Multi-Tessellation

In this Section, we describe a new compact data structure for encoding a Multi-Tessellation (MT) generated through vertex insertion, or vertex removal, that we have proposed in [DDFMP02].

A modification $u = (u^-, u^+)$ based on *vertex insertion* adds a new vertex v_u to a mesh Σ by deleting a set of tetrahedra u^- bounded by a star-shaped polyhedron π_u, and replacing them with the tetrahedra obtained by connecting vertex v_u with all the vertices of π_u (see Figure 5.1 from left to right), which define u^+. The inverse modification, called a *vertex removal*, removes all the tetrahedra incident at a vertex v_u, which define u^+ leaving a hole in the mesh bounded by a polyhedron π_u, and replaces them with a set u^- of tetrahedra obtained by re-triangulating π_u (see Figure 5.1 from right to left).

Criteria to define the tetrahedra forming u^- in a vertex insertion can be the *Delaunay criterion*, in which the tetrahedra in u^- are the subset of the tetrahedra of Σ whose circumsphere contains vertex v_u, or the *half-edge collapse criterion*, in which the tetrahedra in u^- are obtained by contacting an edge e_u of Σ incident at v_u into its other extreme vertex.

A *vertex-based* 3D MT is a 3D MT in which each modification $u = (u^-, u^+)$ describes a vertex insertion, or removal. The nodes of a vertex-based MT are in one-to-one correspondence with the vertices of the reference mesh, except for the vertices of the base mesh, which usually are a negligible amount. Thus, we identify a node u with the vertex v_u inserted/removed by u.

We consider a version of a vertex-based MT built through vertex decimation under the constraint that a vertex can be deleted only if the number of its adjacent vertices is not larger than a predefined constant. We assume that this constant is 32. This constraint is easy to be achieved since, on average, a vertex-removal operation removes less than 19 tetrahedra, that corresponds to 12 vertices adjacent to the removed vertex. In Subsection 5.1.1,

we describe the encoding of the dependency relation among modifications. In Subsection 5.1.2, we describe the encoding of the modifications corresponding to a vertex-insertion operation.

5.1.1 An encoding for the dependency relation

For encoding the dependency relation among modifications, we use a technique proposed by Klein and Gumhold [KG98], described in Subsection 4.5.1. Here, we briefly recall such technique.

Given an MT $\mathcal{M} = (\mathcal{U}, \prec)$, each modification in $\mathcal{U}$ is linked to its parents in the DAG describing relation $\prec$, through a linked cyclic list, called a *loop*. In this way, if a modification has k parents, the number of pointers required to encode the dependency relation is reduced from $2k$ to $k + 1$. An explicit encoding of the dependency relation stores a pointer to and from each parent, while the linked list in the compact encoding contains k pointers plus a pointer to close the loop. Each link encodes the pointer to the next modification and the index of the next link of the loop.

If we consider all modifications in the MT, the total number of links to describe the loops is equal to the number of direct dependencies, which is the number of arcs of the DAG, plus a link for each loop to close it. The number of loops is equal to the number of modifications in the MT. The total number of links is equal to $a + c$, where a is the number of arcs in the DAG, and c is the number of modifications.

We give below a pseudo-code description of the implementation of primitive PARENTS and CHILDREN on the compact DAG encoding. In the algorithm description, we use the following notation:

- given a modification u, we denote with u.link[0] the link to the first parent of u. Each link is a pair composed of a modification and the index to the next link;
- given a link (u', i), we denote with u'.link[i] the link to the next parent in the same loop. Note that$(u', -1)$ indicates that we have reached the last modification of the loop;

Evaluation of the storage cost Each link consists of a pointer to a node plus a loop identifier. We assume that a modification has at most 32 parents or children. This is controlled by the refinement, or the coarsening algorithm. One Byte is sufficient to encode a loop identifier as well as to count the loops to which a node belongs. Experimentally, it has also been found that, for a 3D vertex-based MT, a is about $5n$ on average and c can

be approximated with n. Thus, the cost of storing the DAG is equal to $5 \cdot (a + c) + c = 31n$ Bytes, on average.

Implementation of the primitives Given a modification u, the list of modifications $u_j, \ldots u_k$ from which u directly depends can be extracted in optimal time just following the loop starting at u. Primitive PARENTS can be implemented through the following pseudo-code:

```
Primitive PARENTS(M, S, Σ_S, u)
begin
   P = empty_set()
/* extract the first parent linked by the loop */
   (u', i) = u.link[0]
/* proceed until the end of the loop */
   while i ≠ −1 do
/* add the modification to the set of parents */
      add_set(P, u')
      (u', i) = u'.link[i]
   endwhile
   return(P)
end PARENTS
```

The complexity of primitive PARENTS is optimal, since it is linear in the number of output elements.

The list of modification $u_j, \ldots u_k$ directly depending on u cannot be extracted in optimal time, since, for each loop passing through u, we can reach only one child of u, which is stored at the end of the loop. In the worst case, this leads to a quadratic cost in the number of modifications, for each loop. The primitive CHILDREN can be implemented through the following pseudo-code:

```
Primitive CHILDREN(M, S, Σ_S, u)
begin
   C = empty_set()
   n = u.number_of_loops
/* check each loop pointing to a child */
   for each i so that 0 < i < n do
      (u', j) = u.link[i]  /* reach the end of the loop */
      while j ≠ −1 do
         (u', j) = u'.link_j()
      endwhile
/* add the modification to the set of children */
      add_set(C, u')
   endfor
   return(C)
end CHILDREN
```

5.1.2 An encoding for the modifications

The idea is to encode an implicit and procedural description of a modification, which contains sufficient information to perform vertex insertion and vertex removal on the currently extracted mesh.

For each modification u, we store:

- a compact encoding of the tetrahedralization of the *region of influence* of vertex insertion, i.e., of u^-;
- coordinates of the vertex v_u introduced by u;
- field value at vertex v_u;
- approximation error associated with the modification.

Since each modification u corresponds to a vertex v_u, modifications and vertices are renumbered in such a way that a node u and its corresponding vertex v_u have the same label. Note that an encoding structure for the topology of u^- must store those information, that are sufficient for performing the following two basic tasks:

1. given the current mesh Σ_S and a vertex insertion u, recognize the tetrahedra forming u^- among those of Σ_S;
2. given polyhedron π_u bounding u^-, build u^-.

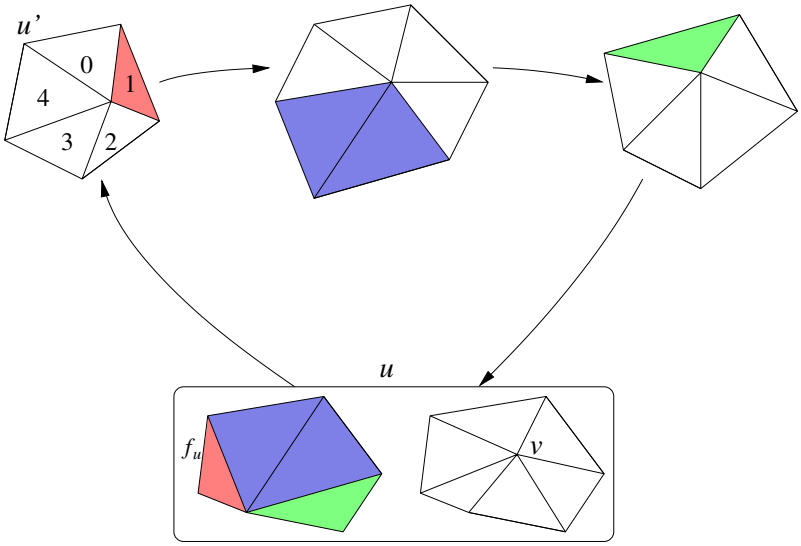


Figure 5.2: Encoding the starting tetrahedron σ_u of modification u: u' is the first parent of u (it is the first modification in the loop). The index of tetrahedron σ_u is encoded. f_u belongs to faces of u' and is encoded in two bits.

Both tasks need encoding one boundary face f_u of the star-shaped polyhedron π_u plus a bit stream which describes a traversal of the tetrahedra of u^-, starting at f_u. A boundary face f_u is described by the tetrahedron σ_u in u^- containing f_u plus the index of f_u within σ_u, and by the tetrahedron σ'_u containing f_u among the tetrahedra incident in v_u after u^+ has been performed.

When performing incremental selective refinement, a tetrahedron σ is generated in the currently extracted mesh Σ_S either during refinement or during coarsening. Thus, a tetrahedron σ is labeled with one bit to discriminate between the two cases, and with an integer, which uniquely identifies σ among the tetrahedra that have been inserted in Σ together with σ.

When a modification $u = (u^-, u^+)$ is applied to Σ_S (by replacing u^+ with u^-), the tetrahedra generated by u are labeled in an order which is always the same every time the modification is applied, and such that the first new tetrahedron is σ'_u. In a similar way, when a modification u is removed from Σ_S (by replacing u^+ with u^-), the tetrahedra generated by the modification are labeled according to an order which is always the same every time, and such that the first new tetrahedron is always σ_u.

Tetrahedron σ_u is identified in a different way depending on whether it has been created in the currently extracted mesh through refinement (i.e., vertex insertion), or coarsening (i.e., vertex removal) step.

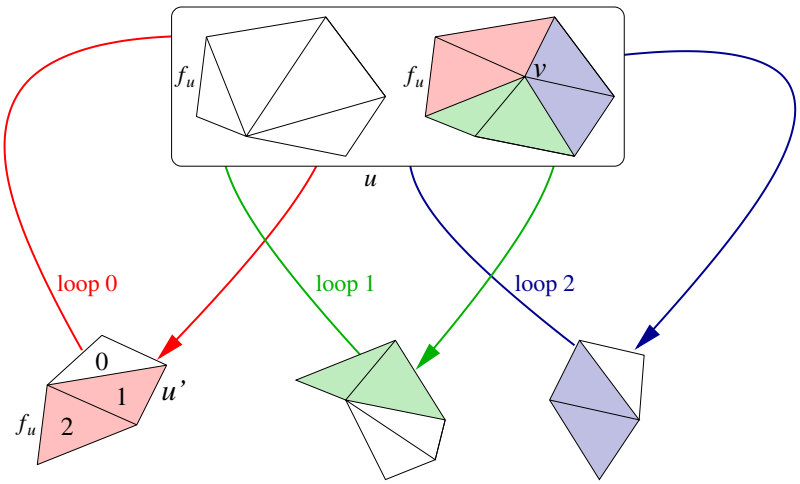


Figure 5.3: Encoding the starting tetrahedron σ'_u of modification u: u' is the first child of u (it is the last modification of the first loop). The index of tetrahedron σ'_u is encoded. f_u belongs to faces of u' and is opposite to the new vertex v_u.

(i) σ_u has been generated by a vertex insertion corresponding to a modification u', which is a direct ancestor of u. The direct ancestors of u are encoded in the DAG in a loop in such a way that u' is the first direct ancestor of u, i.e., the link u.link[0] refers to u'. We just need to identify σ_u among tetrahedra in u'. This can be done with an index. Since the number of vertices in u' is bounded by construction to 32, the number of tetrahedra is not greater than 60 and, thus, the index can be encoded in six bits.

(ii) σ_u has been generated by a vertex removal of vertex v_u. In this case, σ_u is just the first tetrahedron in the set of tetrahedra affected by u.

In a completely symmetric way, tetrahedron σ'_u is identified in a different way depending on whether it has been created in the currently extracted mesh through a vertex removal or a vertex insertion:

(iii) σ'_u has been generated by a vertex removal corresponding to a modification u' which is a direct descendant of u. Also, the direct descendants of u are encoded in such a way that u' is the first direct descendant of u, i.e., it belongs to the first loop ending in a child of u. We just need to identify σ'_u among tetrahedra in u'. This can be done with an index. Since the number of vertices in u' is bounded by construction to 32, the number of tetrahedra is not greater than 60 and, thus, the index can be encoded in 6 bits.

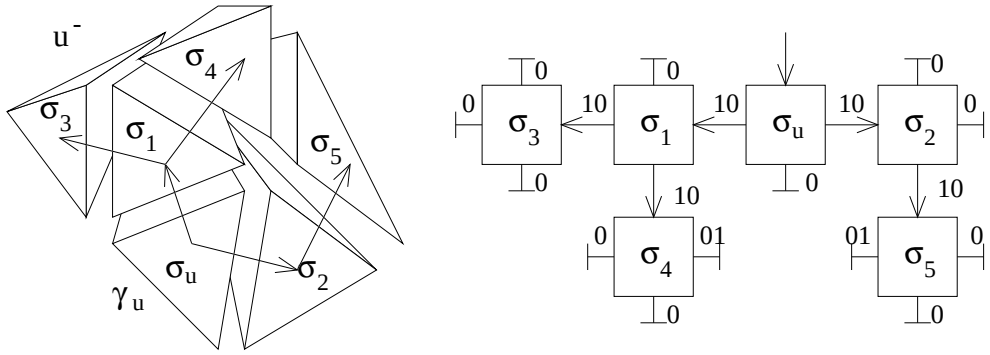


Figure 5.4: The tetrahedron spanning tree. On the left, region of influence u^- of the modifications is shown along with the starting face f_u, and the spanning tree. On the right, the binary labels corresponding to each arc of the spanning tree are shown.

(iv) σ'_u has been generated by insertion of vertex v_u. In this case, σ'_u is just the first tetrahedron incident into v_u.

Once we have identified σ_u, we need two extra bits to identify f_u among the faces of σ_u. Given σ'_u, face f_u is the face opposite to v_u and is encoded at null cost. The total cost is 14 bits per modification.

The tetrahedra of u^- are described as a bit stream that encodes a *tetrahedron spanning tree* rooted at σ_u. The nodes of such tree correspond to the tetrahedra in u^-, while its arcs correspond to some of the faces shared by such tetrahedra. The tetrahedron spanning tree is constructed through a depth-first traversal (while building the LOD model) as follows. Starting from σ_u, all tetrahedra of u^- are traversed in a depth-first fashion. Each triangular face f of a tetrahedron traversed is labeled (see Figure 5.4):

- 0, if f is a face of polyhedron π_u;
- 10, if f is incident into a tetrahedron that belongs to u^- and has not yet been traversed;
- 11, if f is incident into a tetrahedron that belongs to u^- and has already been traversed.

Note that exactly three faces are labeled for each tetrahedron.

Evaluation of the storage cost If u^- contains p tetrahedra and has q external faces, then the bit stream contains $6p - q + 1$ bits encoding u^-. Note that the number q of external faces is equal to the number of new tetrahedra created by vertex insertion, i.e.,

in u^+. Summing up over all nodes we have $6\sum_u p - \sum_u q + m$ where $\sum_u q$ is the total number of tetrahedra stored in the multiresolution model and $\sum_u p$ is the total number of tetrahedra stored in the model minus the number of tetrahedra in the reference mesh. Since, on average, we have that $p \simeq 13$ and $q \simeq 18.6$, the average length of the bitstream is 60.4 bits. Thus, the cost for storing all the modifications is equal to 74.4 n bits ($< 10\ n$ Bytes). Vertex coordinates, field values and error values are stored separately, and require two Bytes each, with a total cost of $10n$ Bytes.

Implementation of the primitives Below, we give a pseudo-code description of the algorithms for implementing primitives REFINE and ABSTRACT. Given a modification u and the current mesh Σ_S, primitive REFINE identifies the starting tetrahedron σ_u among the tetrahedra in Σ_S, and then the starting face f_u. Note that the current mesh Σ_S is encoded in an extended indexed data-structure with adjacency (see Section 2.2.3). Recall that the extended indexed data-structure with adjacencies encodes, for each tetrahedron σ, $R_{30}(\sigma)$ and $R_{33}(\sigma)$, i.e. the indexes of the four vertices of σ and of the four tetrahedra face-adjacent to σ, and, for each vertex v, partial $R^*_{03}(v)$ relation, which encodes the index of just one tetrahedron incident in v. Then, it performs the visit of the tetrahedra of u^- according to the bitstream. It stores in a list the faces of polygon π_u. Once vertex v_u is added to Σ_S, all faces in π_u are connected to v_u, thus creating a set of new tetrahedra. Such operations require the adjacency relation among tetrahedra in the current mesh. Primitive REFINE can be implemented with following pseudo-code:

```
Primitive REFINE(M, S, Σ_S, u)
begin
/* retrieve starting tetrahedron and starting face */
   σ = u.starting_tetra()
   f_u = u.starting_face(σ)
/* load bitstream and new vertex v_u */
   B = u.bitstream()
   v = u.new_vertex()
   for each face f' of σ different from f_u do
/* start recursive visit */
      REFINE_AUX(Σ_S, B, f', v)
   endfor
/* tetrahedra to be removed are marked by procedure REFINE_AUX */
   remove_marked_tetrahedra()
/* adjacencies between tetrahedra in  u+ and between this latter */
/* with the tetrahedra of Σ_S bounding π_u are updated */
   adjust_adjacencies()
end REFINE
```

```
Procedure REFINE_AUX(Σ_S, B, f, v)
begin
   if next_bit(B) = 0 then
/* case 0: boundary face - replace old tetrahedron with a new one */
      let f the i-th face of tetrahedron σ
      σ' =make_new_tetra(f, v)
      add(Σ_S, σ')
      mark_to_remove(Σ_S, σ)
   else if next_bit(B) = 0 then
/* case 10: internal face - not yet visited */
      let f the i-th face of tetrahedron σ
      for each face f' of σ different from f do
      REFINE_AUX(Σ_S, B, f', v)
      endfor
   else
/* case 11: internal face - already visited - nothing to do */
end REFINE_AUX
```

The time complexity of primitive REFINE is linear in the number of tetrahedra in u^+. The algorithm starts from the first tetrahedron and performs a visit of the tetrahedron spanning tree. The number of steps in the above algorithm is linear into the number of faces of such tetrahedra. During the visit, it stores a triangular mesh representing the boundary of u^-. Then, it deletes the old tetrahedra and creates the new ones. Each deletion or creation can be done in constant time. If the 2D mesh representing the boundary faces is encoded by using an extended indexed data-structure with adjacencies, the computation of adjacency relation between tetrahedra in u^+ can be done in linear time. Otherwise, adjacency relation need to be computed from scratch which results in a cost in the order of $O(q\ logq)$, where q is the number of tetrahedra in u^+.

Given a modification u and the current mesh Σ_S, primitive ABSTRACT identifies the starting tetrahedron σ among tetrahedra in Σ_S, and then the starting face f_u. Also, here the current mesh Σ_S is encoded as an extended indexed data structure with adjacencies. Then, it performs the traversal of the tetrahedra of u^+ by computing relation $R_{03}(v)$ for each vertex v (we call it Vertex-Tetrahedra relation). Such tetrahedra are marked and later removed. The simplicial mesh u^- is constructed in three steps: the first step creates a tetrahedron spanning tree with a label of the faces of the tetrahedra as boundary or internal. The second step starts from the root of the tree and glues internal faces. The third step starts from the starting face f_u on the boundary π_u of the modification and matches π_u with the boundary faces of the tetrahedron spanning tree. A pseudo-code description of primitive ABSTRACT is given below.

```
Primitive ABSTRACT(M, S, Σ_S, u)
begin
/* retrieve starting tetrahedron and starting face */
   σ = u.starting_tetra()
   f_u = u.starting_face(σ)
   B = u.bitstream()
   v = u.new_vertex()
/* initialize the mesh representing u⁻ and its boundary */
   M = empty_mesh()
   F = empty_face_set()
/* create the tetrahedron spanning tree */
   ABSTRACT_AUX(M, B, f_u)
/* retrieve the tetrahedra to be removed and modification boundary */
   T = get_Vertex_Triangle(Σ_S, v, σ)
   for each σ' ∈ T do
      let f' be the face of σ' opposite to v
      add_set(F, f')
      mark_to_remove(Σ_S, σ')
   endfor
   glue_internal_faces(M)
/* match external faces of M and faces of the hole created removing u⁺ */
   match_boundary_faces(M, F)
   remove_marked_tetrahedra()
end ABSTRACT
```

```
Procedure ABSTRACT_AUX(M, B, f)
begin
   if next_bit(B) = 0 then
/* case 0: boundary face - nothing to do */
      mark f as boundary face
   else if next_bit(B) = 0 then
/* case 10: internal face - expand tetrahedra tree */
      σ' = new_tetra()
      for each face f' of σ different from f do
         ABSTRACT_AUX(M, B, f')
      endfor
   else
/* case 11: internal face - nothing to do */
      mark f as internal face
end ABSTRACT_AUX
```

The time complexity of primitive ABSTRACT is linear in the number of tetrahedra in u^-: it starts from the new tetrahedron and creates the tetrahedron spanning tree. Each step for

the construction of the tetrahedron spanning tree can be performed in constant time, and the number of steps is equal to the number of triangular faces of u^-. During this process internal faces (to be glued) and the boundary faces are also stored. The computation of relation R_{03} is linear in the number of tetrahedra in u^+. The gluing step is linear in the number of faces that lie on the boundary of the tetrahedron spanning tree. The last step of the construction process finds the matching between the boundary of the hole created by removing u^+ and the boundary of the tetrahedron spanning tree just glued. This step is linear in the number of such faces, since it is performed through to a simple traversal of both surfaces.

5.1.3 Storage costs of the data structure

We can evaluate the storage cost of a vertex-based 3D MT by summing up the cost for the encoding of the dependency relation, information for retrieving the starting tetrahedron, the bitstream and geometric information. This gives a total space required of about $51n$ Bytes. Thus, the data structure achieves a compression factor of up to 6.1 with respect to the explicit data structure for encoding an MT (see Section 4.4). Also, it requires about 52.7% of the space needed by an indexed structure storing the reference mesh and 28% of the space needed by an indexed data structure with adjacencies storing the reference mesh.

5.1.4 Analysis

Building a multiresolution model requires a coarse representation of the dataset and a sequence of modifications. A data structure designed for vertex-insertion modifications can be built through a sequence of vertex-insertion/removal, or through half-edge collapse/vertex split.

Only a few techniques have been proposed for performing vertex insertion/removal in a tetrahedral mesh. Some of them [HC94, CDFM+94, GLE97] are based on a refinement strategy, while in [RO96] a mesh decimation algorithm, based on vertex removal, is proposed. Vertex removal cannot be always performed on a tetrahedral mesh. Removing a vertex from a tetrahedral mesh gives rise to a star-shaped polyhedron, which bounds the set of removed tetrahedra. It has been shown that it is NP-hard to tetrahedralize a star shaped polyhedron. Renze and Olivier [RO96] just remove only points such that their region of influence can be tetrahedral with adding other points (usually called *Steiner points*). Refinement techniques work only for meshes with a convex domain. This is due to the intrinsic difficulties in inserting a point in a continued 3D Delaunay triangulation (see [She98]).

Please refer to Section 3.1.2 for a more detailed review of simplification techniques for three-dimensional scalar fields.

Multiresolution models based on vertex-insertion/removal are characterized by small modifications: the number of tetrahedra affected by a modification is usually rather small, i.e., 12 to 13 tetrahedra on average. This results in a high expressive power and good selectivity in case of queries based on a region of interest. As a consequence, the number of tetrahedra necessary to achieve a given approximation quality with a vertex-based MT is usually smaller than those necessary to achieve the same quality with other data structures, such as the one based on full-edge collapse. (see Section 5.5).

5.2 Half-Edge Multi-Tessellation

In this Section, we describe a new compact data structure for a Multi-Tessellation based on half-edge collapse, that we call a *Half-Edge Multi-Tessellation* (*Half-Edge MT*). A half-edge MT can be considered a specialized version of the vertex-based MT that we have proposed in [DDF02]. Half-edge collapse techniques can to handle arbitrary meshes and offer excellent results in term of mesh accuracy. Moreover, Lindstrom and Turk demonstrated in [LT99] that half-edge collapse offers better results than full-edge collapse with collapse in the midpoint. We based our tests on a simplification techniques proposed by Cignoni et al. in [CCM+00], that performs half-edge collapse with an accurate error evaluation. A Half-Edge MT offers better compression ratios and it is easier to query, but it is less flexible with respect to a vertex-based 3D MT, since it can encode only multiresolution model with modifications based on half-edge collapse, and not a generic vertex insertion policy. The encoding of the dependency relation in a Half-Edge MT is the same.

A half-edge collapse operation collapses an edge $e_u = (v, w)$ into one of its extreme vertices, say w. Tetrahedra incident in both v and w are collapsed into triangles, while tetrahedra incident in w are modified by replacing instances of vertex v with vertex w (see Figure 5.5 from left to right). The inverse operation, called a *half-vertex split*, splits a vertex w into a pair of vertices v and w. This modification introduces a fan of tetrahedra incident into the new edge $e_u = (v, w)$ and modifies a subset of tetrahedra incident into w (see Figure 5.18 from right to left). A half-vertex split can also be seen as a special case of vertex insertion, in which the tetrahedralization of u^- is fixed, since each tetrahedron in u^- is incident into vertex w.

A modification based on half-vertex split cannot be encoded with the data structure designed for full-vertex split [CDFM+04]. A full-vertex split of vertex w affects all simplices incident into vertex w, while a half-vertex split affects only a subset of them. For encoding the dependency relation among modifications, we use a technique proposed by Klein and Gumhold [KG98] described in detail in Subsection 4.5.1. On the other hand, a half-edge

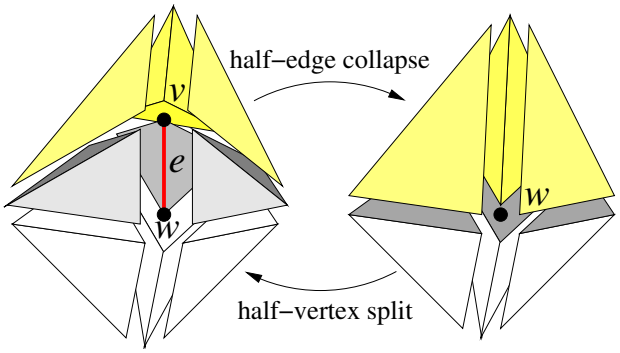


Figure 5.5: Half-edge collapse and vertex split on a tetrahedral mesh (exploded view). On the left, the set of tetrahedra incident into edge e is gray shaded. Such tetrahedra are collapsed to triangles. Yellow tetrahedra are affected by the edge collapse On the right, the vertex w and the new tetrahedra incident in w are shown.

collapse is a special case of a vertex-removal, and thus we can use the vertex-based 3D Mt. We have developed a specific data structure, called a *Half-Edge MT*, with a more compact encoding for modifications, specific for half-edge collapse/half-vertex split.

5.2.1 An encoding for the modifications

The encoding of a modification u requires storing information for performing half-vertex splits and half-edge collapses. Let us assume that the half-edge collapse consists of collapsing an edge (v, w) into vertex w, and thus the half-edge split consists of splitting a vertex w into an edge (v, w). For each modification u we store:

- the coordinates of the new vertex v;
- the field value associated to vertex v;
- an error value, $\varepsilon(u)$, which provides an estimate of the approximation error associated with u and is computed as the maximum of the errors associated with the tetrahedra forming u^-.

Note that we usually store the error associated with an update and not with each tetrahedron forming it to obtain a more compact representation. We also store:

- the region of influence u of the modification, bounded by the star-shaped polyhedron π_u;

- a compact encoding of the topological structure of u^-, which is a portion of the star of w;
- and an implicit encoding of vertex w.

To perform a half-edge collapse, we need to encode the vertex w on which edge $e = (v, w)$ is contracted. Since a modification u corresponds to the insertion of a vertex v, modifications and vertices are re-numbered in such a way that a modification u and its corresponding vertex v have the same label. Thus, the relation between u and v is encoded at a null cost.

An encoding structure for the topology of u^- must store those information, that are sufficient for performing the following two basic tasks:

1. given the current mesh Σ_S and a vertex split, recognize the tetrahedra forming u^- among those of Σ_S;
2. given polyhedron π_u bounding u^-, build u^-.

Both tasks need encoding one boundary face f_u of the star-shaped polyhedron π_u plus a bit stream which describes a traversal of the tetrahedra of u^-, starting at f_u.

A boundary face f_u is described by the tetrahedron σ_u in u^- containing f_u plus the index of f_u within σ_u, and by the tetrahedron σ'_u containing f_u among the tetrahedra incident in v_u after u^+ has been performed. We have adopted the same technique described in Subsection 5.1.2.

Modifications based on half-vertex split are much simpler than modifications based on vertex insertion. The tetrahedralization of u^- consists in a fan of tetrahedra incident into a vertex w. We can exploit this property for encoding a simpler traversal.

The tetrahedra of u^- are described as a bitstream that encodes a *tetrahedron spanning tree* rooted at σ_u, which is constructed as follows. Starting from σ_u, all tetrahedra of u^- are traversed in a depth-first fashion. Each triangular face incident into vertex w of a traversed tetrahedron is labeled with

- 0, if the face is on the boundary π_u of u^-;
- 1, if the face is adjacent to a tetrahedron that belongs to u^-.

Note that only two faces are labeled for each tetrahedron. This is also true for the initial tetrahedron σ_u, since we already know that one of its faces, f_u, is a boundary face.

Evaluation of the storage cost If u^- contains p tetrahedra, then the bit stream contains $2p$ bits, since the length of the bit stream does not need to be stored. Since, on average, we have that $p \simeq 13$, the average length of the bitstream is 26 bits.

Tetrahedra σ_u and σ'_u are encoded with 10 bits, while the information to identify face f_u among faces of σ_u, and vertex w among vertices of σ'_u require two bits each. This results in an average cost of 40 bits per modification that sums up to $5n$ Bytes (compared with the $10n$ Bytes required by encoding of a vertex insertion/removal). Vertex coordinates, field values and error value are stored separately, and require two Bytes each, with a total cost of $10n$ Bytes. Thus, the storage cost for the information associated with all modifications contributes for a cost of $15n$ Bytes.

Implementation of the primitives Given a modification u and the current mesh Σ_S, primitive REFINE identifies the starting tetrahedron σ_u among tetrahedra in Σ_S, and then the starting face f_u. Then, it performs the traversal of the tetrahedra contained in u^- according to the encoded bitstream. Vertex v is added to Σ_S, all faces belonging to π_u and incident into vertex w are split into new tetrahedra, resulting in a fan of tetrahedra incident into the new edge $e = (v, w)$. Tetrahedra of u^- are modified by replacing vertex w with new vertex v. Previous operations require that the current mesh Σ_S is encoded as an extended indexed data structure with adjacencies. The pseudo-code description of primitive REFINE is reported below:

```
Primitive REFINE(M, S, Σ_S, u)
begin
/* retrieve starting tetrahedron and starting face */
   σ = u.starting_tetra()
   f_u = u.starting_face(σ)
/* load bitstream and new vertex v_u */
   B = u.bitstream()
   v = u.new_vertex()
   for each face f' of σ incident in w different from f_u do
/* start recursive visit */
      REFINE_AUX(Σ_S, B, f', v)
/* compute the adjacencies among new tetrahedra */
   adjust_adjacencies()
end REFINE
```

```
Procedure REFINE_AUX(Σ_S, B, f, v)
begin
   if next_bit(B) = 0 then
/* case 0: boundary face incident in w - add new tetrahedron */
      σ' =make_new_tetra(f, v)
/* set tetrahedra adjacent to f as adjacent to σ' */
      adjust_adjacencies(σ', f)
      add(Σ_S, σ')
   else
/* case 1: internal face - modify existing tetrahedron */
      Let f be the face on tetrahedron σ
      replace_vertex(σ, w, v)
      for each face f' of σ incident in w different from f do
        REFINE_AUX(Σ_S, B, f', v)
      endfor
end REFINE_AUX
```

The time complexity of primitive REFINE belongs to $O(p+q \cdot logq)$ where p in the number of tetrahedra in u^-, while q is the number of new tetrahedra. The process starts from the first tetrahedron and performs a visit of the tetrahedron spanning tree. The number of steps is linear into the number of faces of such tetrahedra. Only faces incident into vertex w are visited. Visited tetrahedra are modified by replacing vertex w with the new vertex v. A new tetrahedron for each face on the boundary of u^- incident in vertex w is created. Adjacencies among new tetrahedra are computed, while adjacencies among new and old tetrahedra are just stored. Computing adjacencies among new tetrahedra means sorting them around edge $e = (v, w)$. This introduces the logarithmic factor in the time complexity. On average, the number of new tetrahedra is five, while the number of tetrahedra in p is about 13, so $p > q \cdot logq$ and the overall behavior is linear.

Given a modification u and the current mesh Σ_S, primitive ABSTRACT identifies the starting tetrahedron σ_u among the tetrahedra in Σ_S, and then vertex w. Than, it traverses the tetrahedra in u^+ and computes relation R_{03}. Note that $R_{03}(w)$ can be computed from partial $R^*_{03}(w)$ encoded in the data structure. Some tetrahedra incident into vertex w are removed while other tetrahedra are modified replacing vertex v with w. At the end, vertex v is removed from Σ_S. A pseudo-code description of primitive ABSTRACT is given below.

```
Primitive ABSTRACT(M, S, Σ_S, u)
begin
/* retrieve starting tetrahedron and vertices v and w */
   σ = u.starting_tetra()
   w = u.old_vertex(σ)
   v = u.new_vertex()
/* retrieve tetrahedra incident into vertex v */
   T = get_Vertex-Tetrahedra(Σ_S, v, σ)
   for each σ' ∈ T do
      if σ' incident into w then
         delete_tetra(σ')
/* update adjacencies of tetrahedra adjacent to σ' and */
/* opposite to v and w: they are glued and σ' is removed.*/
         update_adjacencies(σ')
      else
         replace_vertex(σ', v, w)
   endfor
end ABSTRACT
```

The time complexity of primitive ABSTRACT is linear in the number of tetrahedra in u^+. Given vertex v, $R_{03}(w)$ is computed in linear time in the number of tetrahedra in u^+. Checking if a tetrahedron σ is incident into vertex w, and removing it, or replacing v with w requires a constant time. Adjacencies can be adjusted in constant time when a tetrahedron is removed and do not have to be computed from scratch. Compared to the primitive ABSTRACT described in Subsection 5.1.2, this primitive is much simpler since it just modifies a vertex of some tetrahedra and delete the others. In case of vertex insertion, we need to remove all tetrahedra in u^+ and then recreate the set of tetrahedra in u^- starting from the tetrahedron spanning tree.

5.2.2 Storage costs of the data structure

The Half-Edge MT, and the vertex-based 3D MT, are characterized by small modifications. On average, the region of influence u^- of a modification u is composed by 13 tetrahedra. This results in high expressive power and good selectivity in case of queries based on a region of interest.

The total cost of a half-edge MT data structure, including the cost of encoding the dependency relation, information for retrieving the starting face, the bitstream and geometric information, is equal to $15n + 31n = 46n$ Bytes.

This structure achieves a compression factor of up to 6.67 with respect to the explicit MT data structure. Also it requires about 49% of the space needed by an indexed structure

storing just the reference mesh, and 26% of the space required for encoding the reference mesh in an indexed data-structure with adjacencies.

5.3 The Half-Edge Tree

In this section, we describe an implicit data structure for an MT built through a half-edge collapse, in which a partial order is encoded as a tree by extending an approach proposed by El-Sana and Varshney [ESV99] for triangle meshes simplified through full-edge collapse. We call this new data structure a *Half-Edge Tree* (HET). Here, the encoding of the dependency relation has been improved, while the encoding of the modification is similar to the technique described in Section 5.2 for a Half-Edge MT.

5.3.1 An encoding of the dependency relation

Here, we describe a compact way to encode a direct dependency relation of MT constructed using an half-edge collapse operation. We adapt the *view dependent tree*, a mechanism proposed by El-Sana and Varshney [ESV99] for implicit encoding of dependency relation. This mechanism is based on a binary forest of vertices, and on a vertex enumeration scheme. This encoding scheme applies to MT built though full-edge collapse that contracts an edge to a new point. We have extended the data structure of El-Sana and Varshney to support half-edge collapses and half-vertex splits applied on tetrahedral meshes.

In the original data structure the leaves of the forest correspond to the vertices of the reference mesh, while other nodes correspond to the vertices generated through full-edge collapse. In particular, the roots correspond to vertices of the base mesh. The two children of each internal node v correspond to the endpoints v' and v" of the edge e created when splitting v. The n vertices of the reference mesh are labeled arbitrarily with numbers from 1 to n. The remaining vertices are assigned consecutive numbers.

Figure 5.6 shows a sequence of full-edge collapses which are applied to the reference mesh and progressively coarsen it to the base mesh (the example is given in two dimensions, for the sake of clarity). Figure 5.7 shows the corresponding binary forest with the enumeration of the vertices in the order of their creation. A sequence of full-vertex splits that progressively refine a triangle base mesh is shown in Figure 5.8.

Given a mesh Σ_S corresponding to a consistent set S of modifications, the following properties hold [ESV99]:

1. a full-vertex split is feasible on Σ_S if and only if vertex v belongs to Σ_S and all the labels of vertices adjacent to v are lower than that of v. If a full-vertex split is not

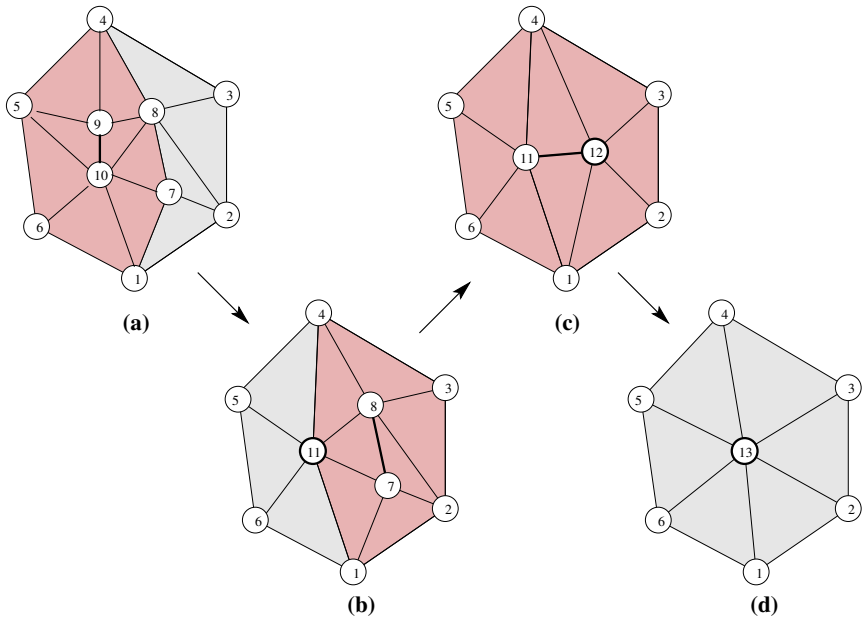


Figure 5.6: Sequence of full-edge collapses on a triangle mesh: (a) edge $(9, 10)$ is collapsed to the new vertex 11, (b) edge $(7, 8)$ is collapsed to the new vertex 12, (c) edge $(11, 12)$ is collapsed to the new vertex 13, (d) the base mesh. In (a), (b) and (c), the red color indicates the triangles affected by the collapse (which are the triangles in the star of the extreme vertices of the collapse edge).

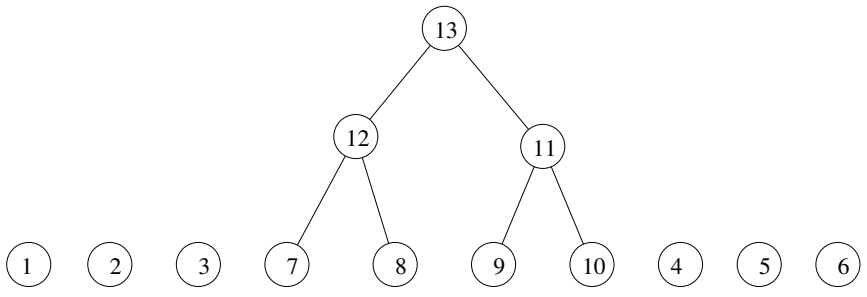


Figure 5.7: A binary forest corresponding to the sequence of full-edge collapses depicted in Figure 5.6.

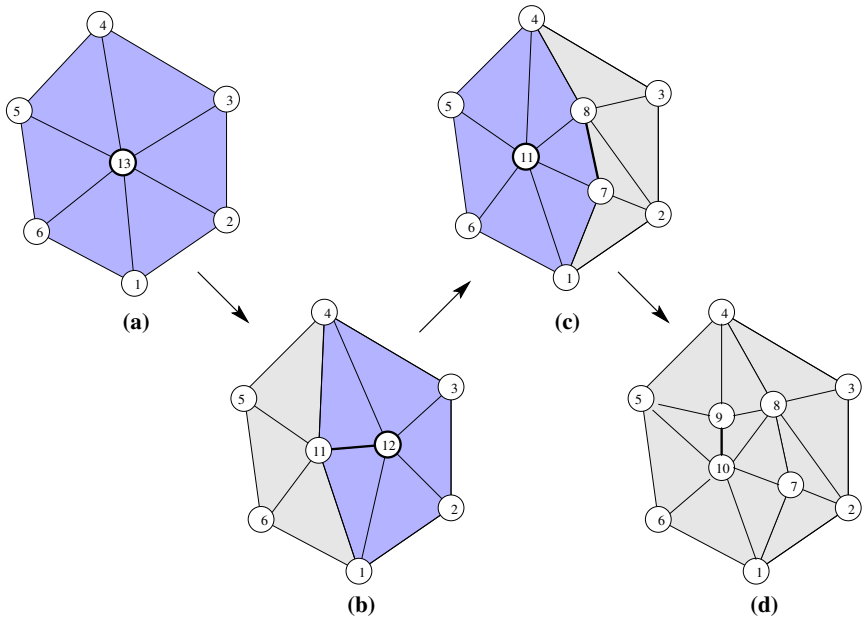


Figure 5.8: Sequence of full-edge splits: (a) vertex 13 is split into the edge $(11, 12)$, (b) vertex 12 is split into the edge $(7, 8)$, (c) vertex 11 is split into the edge $(9, 10)$, (d) the reference mesh (mesh of full resolution). In (a), (b) and (c) the blue color indicates the triangles affected by the full-edge split (that all those triangles in the star of a vertex to be split).

feasible, then it can be made feasible by first splitting all the adjacent vertices of v which have a label greater than v. If any of such adjacent splits is not feasible, the rule is recursively applied;

2. a full-edge collapse is feasible on Σ_S if and only if edge e belongs to Σ_S and, for each vertex w adjacent to the endpoints of e in Σ_S, either w is one of the roots in the forest, or the label of the parent of w is greater than the label of vertex created by collapsing e.

We have modified the new dependent tree as described below.

Since a half-edge collapse does not create a new vertex, we rename one of the endpoints of the edge and consider it as another vertex. For example, if edge (v, w) collapses to vertex w, then w' is a renamed copy of vertex w and in the binary tree is stored as the parent of vertices v and w. By convention, vertex w is a left son of w' and called the *false* son of w', while vertex v is a right son of w' and called the *true* son of w'. Accordingly, w' is the *false* parent of w and the *true* parent of v (Figure 5.9).

The difference between half-edge collapse and full-edge collapse does not affect the vertex

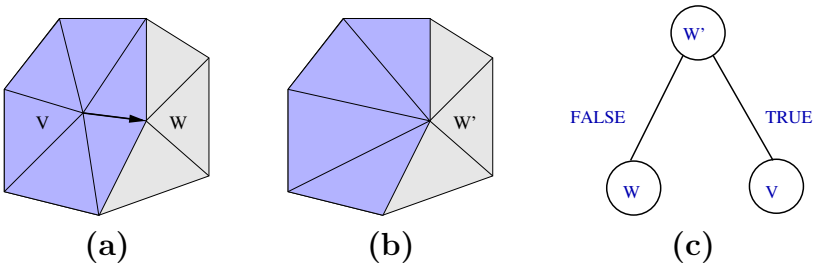


Figure 5.9: From (a) to (b): half-edge collapse. Note that w' is the same vertex as w but with a different label. From (b) to (a): half-vertex split operation. The polygon affected by updates is colored in blue. (c) Binary tree describing the half-edge vertex split updates. The left child is called the false-child of w' since it describes the same vertex as w'.

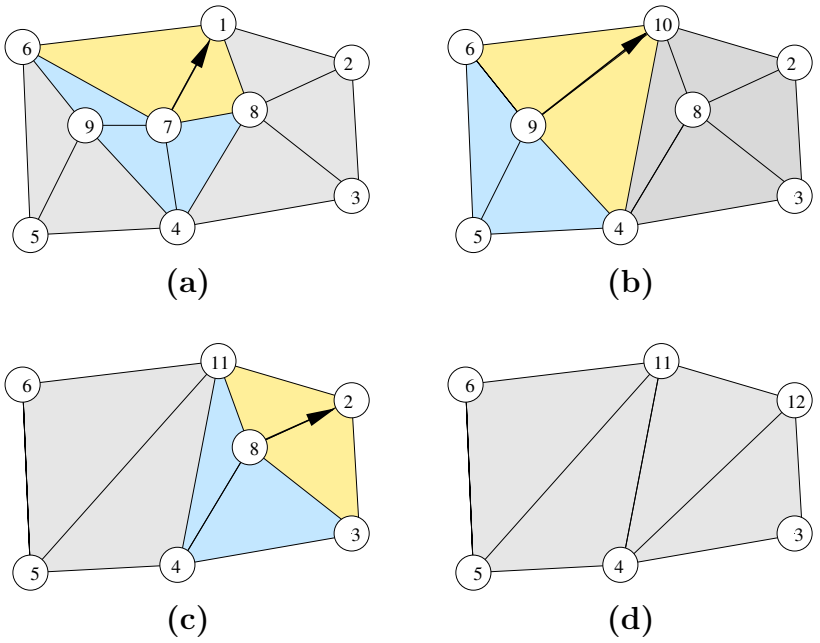


Figure 5.10: Sequence of half-edge collapses: (a) edge $(7, 1)$ is collapsed to vertex 1 that is labeled 10, (b) edge $(9, 10)$ is collapsed to vertex 10 that is labeled 11, (c) edge $(8, 2)$ is collapsed to vertex 2 that is labeled 12, (d) the base mesh. The triangles that removed as the result of the collapse operation are drawn in yellow, the triangles that are updated are shown in blue.

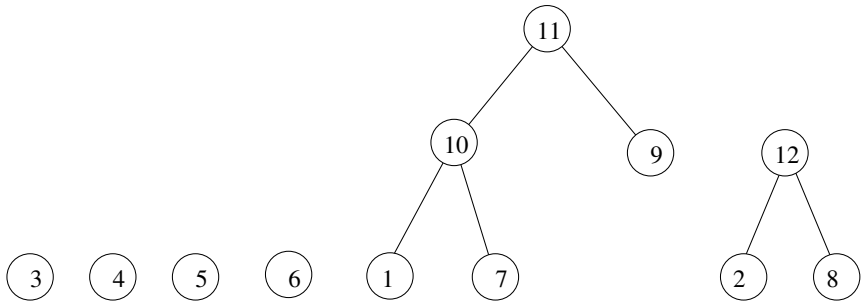


Figure 5.11: The binary forest correspond to half-edge collapse sequence depicted in Figure 5.10.

enumeration mechanism. Figure 5.10 shows a sequence of half-edge collapses and the corresponding vertex labels. Figure 5.11 shows the forest of view dependent trees corresponding to the half-edge collapses depicted in Figure 5.10.

Feasibility test has to be modified for this extended view dependent tree. We call a *neighbor* of vertex v a vertex in the $star(v)$. Given a modification u based on half vertex split, we call *relevant neighbors* of vertex w the vertices (different from w) of the tetrahedra of the region of influence of u.

The true parent of v is the true parent of the nearest $\overline{v}$ ancestor of v, such that $\overline{v}$ is a true son. The true parent of v may not exist. In this case vertex v is the result of iterated renaming of a vertex of the base mesh. See the links for true parents (if they exist) at Figure 5.12 (for false sons these links are shown by dashed lines).

The definition of a true parent of node v can be formulated as follows:

$$\textit{true parent(v)} \text{ is } \begin{cases} \textit{parent(v)} & \text{if } v \text{ is a right son} \\ \textit{true parent(u)} & \text{if } v \text{ is a left son, where} \\ & u \text{ is the nearest ancestor of } v, \text{ and} \\ & u \text{ is a right son} \\ \textit{null} & \text{if } v \text{ is a vertex of a base mash} \end{cases}$$

We define here the rules for splitting and collapsing based on the extended new dependent tree.

Rules for splitting Let vertex w be the vertex split by modification u. Vertex w can be split if all modifications u', such that $u' \prec u$, are already in S. That means, that each vertex v, a relevant neighbor of w, has a label lower than w. This rule and its proof are the same as for full-edge split, except that we refer only to the relevant neighbors.

In order to prove the correctness of rule, we have to show that all modifications u', such

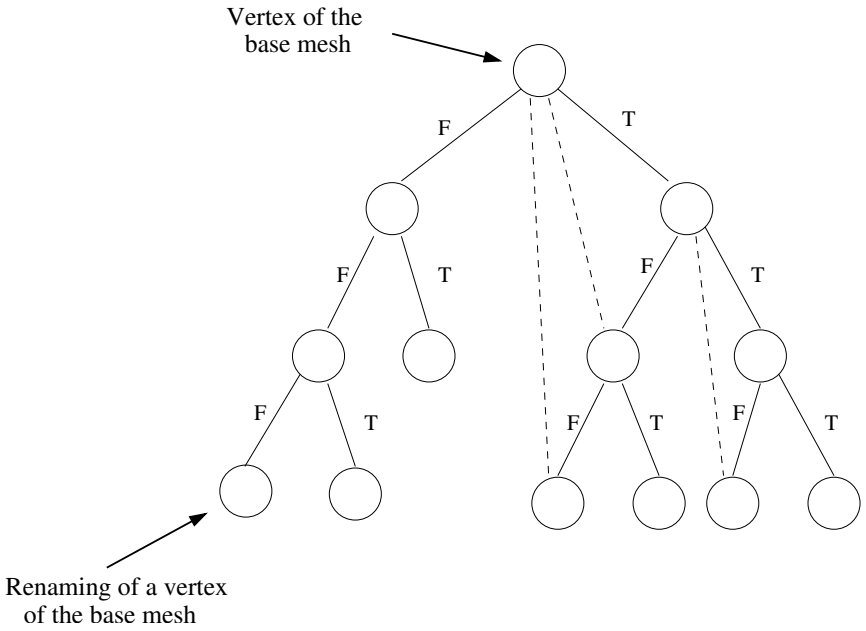


Figure 5.12: The right child of the node in binary tree is the true-child, while the left child is the false-child. Parent is a renamed copy of the left child. true parents of the left children (if they exist) are shown by dashed lines.

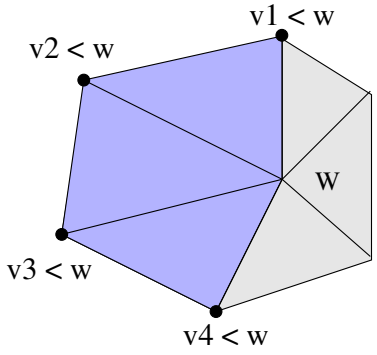


Figure 5.13: Labels of relevant neighbors of w should be less than label of w in order to perform split operation.

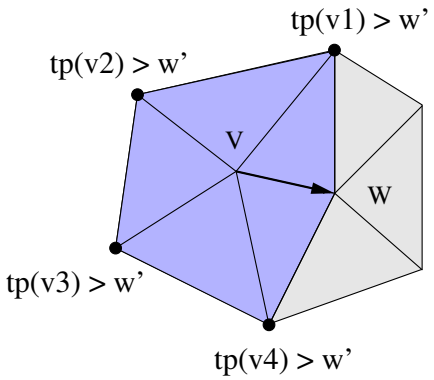


Figure 5.14: In order to collapse the edge (v, w) into vertex w', the labels of the true parents of neighbors of v should be larger than label of w' or no true parent. Note that $tp(v)$ denotes the true parent of vertex v.

that $u' \prec u$, are in S if and only if each vertex v, which is a relevant neighbor of w, has a lower label than v.

1. If some modification u', such that $u' \prec u$, is not in S, then some relevant neighbor of w has a label greater than w. This is true because if u' is not performed and $u' \prec u$, then $v_{u'}$ is greater than w, and $v_{u'}$ must be a neighbor of w because u and u' are related in the partial order.

2. If some relevant neighbor v of vertex w has greater label, then there exists update u', such that u' splits v. Since w and v are neighbors, then u and u' must be related in the partial order, therefore $u' \prec u$, and u' is not in S.

Rules for collapsing An edge (w, v) can be collapsed to a vertex w' if and only if no modifications u', such that $u \prec u'$, is in S. This means, that all vertices adjacent to vertex v (except of w) have true parent with a label greater than label of w', or no true parent (see Figure 5.14).

In order to prove the correctness of this rule, we have to show that no modifications u', such that $u \prec u'$, is in S if and only if each neighbor vertex of v has a true parent with greater label than w'.

1. If some modifications u', such that $u \prec u'$, is in S, then the label of $v_{u'}$ is lower than the label of w'. One vertex at the current mesh has $v_{u'}$ as its true parent. This vertex must be a neighbor of v, because otherwise there would be no dependency between these modifications.

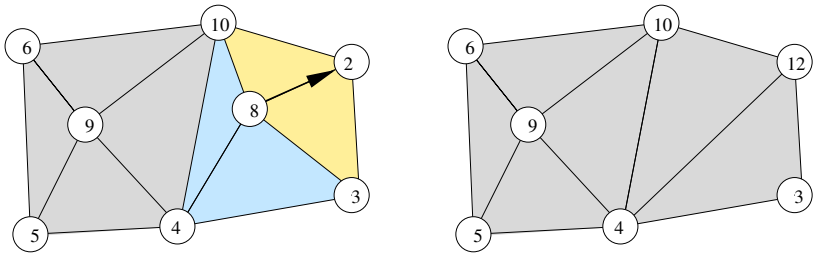


Figure 5.15: Applying legal collapse update of edge $(8, 2)$ on the mesh at step (b).

2. Suppose some vertex v is not a root and has true parent $v_{u'}$ with label lower than w'. Since w and v are neighbors, then modification u and u' must be related in the partial order. In addition, the label of $v_{u'}$ is smaller than a label of w', therefore $u \prec u'$.

The example in Figures 5.10 and 5.11 demonstrates the use of true parents while half-edge collapse updates are performed. Note that the rule for split is the same as for full-edge split modification, but a half-edge collapse of the edge (v, w) into vertex w' can be performed only when labels of true parents of relevant neighbors of v are larger than label of w' or there is no true parent. Suppose we want to perform a collapse of an edge $(8, 2)$ on the mesh Figure 5.10 produced at step (b). We check the true parents of neighbors of vertex 8: vertices 3, 4, 10. All of them have no true parent. Thus, the modification is legal. This is true, since vertices 10 and 11 correspond to the same vertex on the mesh. The result of such modification is shown in Figure 5.15. Note, that if we use the rule for full-edge collapse, this latter modification would be illegal, since the parent of vertex 10 is 11, and it is not greater than the label of new vertex 12. However, if we want to perform the same modification on the mesh in Figure 5.10 produced at step (a), we are not allowed, because the true parent of its neighbor vertex 7 has label 10, that is less than a label of new vertex 12. The result of applying such modification is shown in Figure 5.16. This would create two triangles which were not in the reference mesh.

Implementation of the extended view dependent tree We have implemented the forest as an array in which every node v is stored at the position corresponding to its label. The entry for v contains:

- an index v.WIDE which points either to the parent p of v, if v is the second child of p, or to the sibling of v, if v is the first child of p;
- a bit that specifying the meaning of v.WIDE: if the bit is set to zero v.WIDE points to the father, otherwise to the sibling;

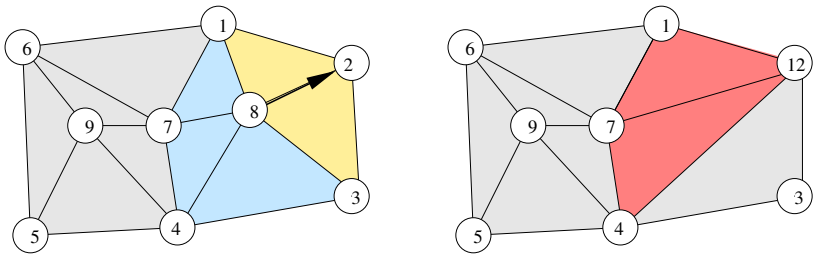


Figure 5.16: Applying illegal collapse update of edge $(8, 2)$ on the mesh of step (a). Illegal triangles are shown in red.

- an index v.DEEP pointing to the first child of v.

In our implementation, we do no store links to a true parent for each false son since it is space consuming. To find a true parent we traverse the tree bottom-up until the first true parent is found. In this way, we reach a true parent without extra space cost, but it is time consuming. We have found experimentally that such path would have a length equal to 2, on average. The combination of two solutions could be to store the pointer only if the number of steps we need to perform to reach a true parent is high.

Evaluation of the storage cost A leaf v of the forest requires just index v.WIDE, while an internal node of forest requires two indexes v.DEEP and v.WIDE and an additional bit. Therefore, our implementation consists of three separate arrays, one for the leaves, one for the internal nodes and one for additional bits. The storage cost of maintaining the forest is equal to $4\ n + 8\ m + 1/8\ m \simeq 12.125n$ Bytes, where n is the number of vertices of the reference mesh (i.e., of leaves in the forest), and m is the number of modification (i.e. internal nodes, and $m \simeq n$).

Implementation of dependency relation: A direct implementation of primitives PARENTS and CHILDREN as in the pseudo-code description in Section 4.3.1 will be very inefficient for a view dependent tree.

Let us consider the example in figure 5.17, which shows a DAG on the left and the corresponding forest of view-dependent trees on the right. Nodes of the view dependent tree represent vertex labels. If we try to compute the parents of modification u_9, we get from the view-dependent tree modification u_3, then, looking for the neighbors of vertex 22, split by modification u_9, we retrieve vertex 25 (modification u_6). Unfortunately u_9 does not directly depend on u_6 so we should apply u_6 compute its neighbors and check if u_9 depends on some of them. We should traverse the view dependent forest recursively, until we get the full set of parents.

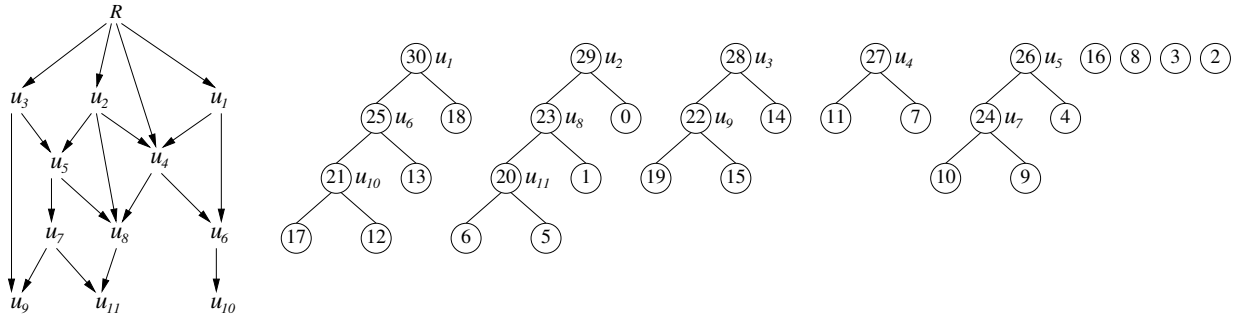


Figure 5.17: A DAG of modifications and the corresponding forest of view-dependent trees.

Since primitives PARENTS and CHILDREN, cannot be implemented efficiently on the view-dependent tree, we have a slightly different implementation of the selective refinement algorithm and of some of the primitives with respect to the one presented in Subsection 4.3.1. We define two primitives for extracting the parents and the children of a modification in the view-dependent tree.

We introduced primitives PARENTS_VDT and CHILDREN_VDT returning only modification directly connected in the view dependent tree. Given a modification u, primitive PARENTS_VDT returns the parents of modification u in a view-dependent tree, while primitive CHILDREN_VDT returns the children of modification u.

```
Primitive PARENTS_VDT(M, u)
begin
   P = empty_set()
   v = u.split_vertex()
/* check if v is first or second child */
   if v.bit = 0 then
      add_set(P, v.WIDE)
   else
      add_set(P, v.WIDE.WIDE)
   return(P)
end PARENTS_VDT
```

```
Primitive CHILDREN_VDT(M, u)
begin
   C = empty_set()
   v = u.split_vertex()
   add_set(C, v.DEEP)
   add_set(C, v.DEEP.WIDE)
   return(C)
end CHILDREN_VDT
```

We define here a new primitive, that we call ANCESTORS. Given a modification u, primitive ANCESTORS returns at each invocation one or more missing ancestors that are required to perform the vertex split encoded in u. In order to retrieve relevant neighbors, each invocation of the ANCESTORS primitive checks the presence of the first tetrahedron in u, checks its vertices and then tries to decode the subset of tetrahedra in u^-: it starts from the first tetrahedron, then visits the mesh according to the implicit encoding of u^+, and it returns one or more vertices split, which have a label greater than w.

```
Primitive ANCESTORS(M, S, Σ_S, u)
begin
   A = empty_set()
   σ = u.starting_tetra()
   w = u.old_vertex()
/* check the presence of σ in Σ_S */
   if σ is not in Σ_S then
/* split σ label in vertex label plus a small index */
      (v', index) = σ
      add_set(A, v')
   else
/* σ is present - check its vertices */
      for each vertex v' of σ different from w do
         if w.label() < v'.label()then
            add_set(A, v')
      endfor
/* if a vertex is missing, stop the procedure and insert it */
   if not_empty(A)
      return(A)

/* perform the test on the next tetrahedron */
   f_u = u.starting_face(σ)
   B = u.bitstream()
   for each face f' of σ incident in w different from f_u do
      A = ANCESTORS_AUX(Σ_S, B, f', w)
   endfor
   return(A)
end ANCESTORS
```

```
Procedure ANCESTORS_AUX(Σ_S, B, f, w)
begin
   A = empty_set()
   (σ, i) = f
/* check vertices of tetrahedron σ */
   for each vertex v' of σ different from w do
      if w.label() < v'.label()then
         add_set(A, v')
   endfor
/* if a vertex is missing, stop the procedure and insert it */
   if not_empty(A)
      return(A)
/* check tetrahedra adjacent to σ according to the bitmask */
   if next_bit(B) == 0 then
/* case 0: boundary face incident in w - nothing to do */
   else
/* case 1: internal face - recursive test */
      for each face f' of σ incident in w different from f do
         A = A∪ ANCESTORS_AUX(Σ_S, B, f', w)
      endfor
   return(A)
end ANCESTORS_AUX
```

Procedure ANCESTORS_AUX traverse tetrahedra belonging to u^- according to the bitsream, and identify vertices in the link of w that need to be splitted since are ancestors of modification u.

The time complexity of primitive ANCESTORS in worst case is linear in the number of tetrahedra in u^-. Each invocation of such primitive returns one or two missing ancestors. Primitive ANCESTORS returns an empty set only if each modification from which u depends is already in S.

Since primitive PARENTS is somehow replaced by primitive ANCESTORS, the definition of primitive FORCE_REFINE that we have used in the selective refinement algorithm in Section 4.3.2 is slightly changed. The new pseudo-code description of primitive FORCE_REFINE is reported below:

```
Procedure FORCE_REFINE(M,S,Σ_S,u)
/* Recursively applies all ancestors of update u that are not in S. */
begin
   A = ANCESTORS(M, S, Σ_S, u)
   while A ≠ ∅ do
      for each u' in A do
         FORCE_REFINE (M,S,Σ_S,u');
      endfor
      A = ANCESTORS(M, S, Σ_S, u)
   endwhile
   add_item(S, u)
   REFINE(M, S, Σ_S, u)
end FORCE_REFINE
```

This implementation strategy, compared to an MT with the dependency relation encoded with a DAG, does not affect time complexity, since every missing ancestor is traversed and added once in both algorithms.

Since the incremental selective refinement algorithm is also affected, instead of primitive CHILDREN, we have here a primitive called FEASIBILITY_COARSEN. Given a modification u, primitive FEASIBILITY_COARSEN checks the feasibility to collapse an edge (v, w) to vertex w renamed in w'. As mentioned before, this modification is allowed only if each vertex adjacent to v is a root in the binary forest, or the label of its true parent is greater than the label of v. Here we assume that (v, w) is an edge of a current mesh Σ_S, and v and w are siblings in view-dependent tree.

```
Primitive FEASIBILITY_COARSEN(M, S, Σ_S, u)
begin
   w' = u.old_vertex()
   v = u.new_vertex()
/* compute the set of vertices adjacent to v */
   LV = Extract_R03(Σ_S, v)
   for each v' in LV do
      v" = TRUE_PARENT(M, S, Σ_S, v')
/* check if v" depends on w', i.e. is a descendant of w' */
      if v".label() < w'.label()then
         return FALSE
   endfor
   return TRUE
end FEASIBILITY_COARSEN
```

The time complexity of primitive FEASIBILITY_COARSEN is in strongly dependent on the complexity of primitive TRUE_PARENT, which returns the true parent of vertex

v. If we store the pointer to the true parent, the time complexity is dominated by the cost of primitive Extract_R_{03} that retrieves vertices adjacent to v, which is linear in the size of the output and thus in the number of tetrahedra of the star of vertex v. If we do not store pointers to true parents, and trees in the binary forest are as high as h, FEASIBILITY_COARSEN can requires up to h operation for each vertex in LV.

5.3.2 An encoding for modifications

The encoding of a modification u requires storing information for performing vertex splits and half-edge collapses. We use a data structure very similar to the one described in Subsection 5.2.1. We do not need to encode vertex w in the collapse of edge $e = (v, w)$ on w.

We encode differently σ_u and σ'_u, since the dependencies are not encoded as a DAG, and thus we do not have an explicit encoding of the relation with the direct ancestors. To encode σ_u and σ'_u we use a labeling technique for tetrahedra. Each tetrahedron σ in the currently extracted mesh Σ_S belongs to the base mesh, or it has been introduced by a modification u. We store a label for tetrahedra in the base mesh. The cost of such indexes in nearly negligible, since base mesh is quite small. If tetrahedron σ has been introduced by modification u, it is labeled with the index of the vertex v_u plus some extra bits. On average, a modification introduces five new tetrahedra and, thus, three bits for each new tetrahedron are sufficient.

Encoding tetrahedron σ_u requires the index of the vertex that introduced it, plus three bits. Two additional bits encode f_u among the faces of σ_u. Tetrahedron σ'_u can be encoded at null cost because its label is composed by vertex v_u plus a few bits, but during the labeling of new tetrahedra, tetrahedron σ'_u is associated to the first index.

The total cost for encoding tetrahedra σ_u, σ'_u and face f_u is thus 4 Bytes for the index of the vertex plus a few bits for tetrahedron σ_u, two bits for face σ_u. This sums up to less than 5 Bytes.

Like in the previous case, we have that the average length of the bitstream is 26 bits. Since encoding σ_u, σ'_u and f_u requires less than 5 Bytes, we have a cost of $9n$ Bytes. Vertex coordinates, field values and error value are stored separately, and require two Bytes each, with a total cost of $10n$ Bytes. Thus, the storage cost for the information associated with a single modification contributes for a cost of less than 19 Bytes.

5.3.3 Storage costs of the data structure

The total cost of a half-edge MT data structure, including the cost of encoding the dependency relation, information for retrieving the starting tetrahedron and face, the bitstream and geometric information, is less than $19n + 13n = 32n$ Bytes.

Data structure presented in this section is more compact that data structure presented in Section 5.2, mainly because of the compact encoding of the dependency relation, but, as a counterpart, implementation and algorithms are more difficult to be implemented.

This structure achieves a compression factor of up to 9.7 with respect to the explicit data structure. It requires 71% of the space of the Half-Edge MT, due to the compact encoding of the dependency relation. Also it requires about 33.8% of the space needed by an indexed structure storing just the reference mesh and 18% of the space needed by a structure with adjacencies.

5.4 Other multiresolution data structures

In this Section, we briefly review two data structures that we compare with the data structures described in the preview sections. Specifically, we compare them versus the *Full-Edge Tree* (FET) proposed in [CDFM$^+$04], which is built through full-edge collapse, and the *Hierarchy of Tetrahedra* (HT) [LDFS01], which is an implicit representation for a multiresolution model based on nested tetrahedral meshes based on tetrahedron bisection. We have performed comparisons versus this representation as well see [DDFLS02a].

To the extent of our knowledge, the FET is the only existing data structure for compact encoding of a multi-resolution model based on unstructured (irregular) tetrahedral meshes.

5.4.1 A Full-Edge Tree

A full-edge collapse operation, collapse an edge $e_u = (v', v")$ into a new vertex v. Tetrahedra incident both in v' and $v"$ are collapsed into triangles, while tetrahedra incident in v' or in $v"$ are modified replacing instances of vertices v' and $v"$ with the new vertex v (see Figure 5.18 from left to right). The inverse operation splits a vertex v into a pair of vertices v' and $v"$. This modification introduces a fun of tetrahedra incident into the new edge $e_u = (v', v")$ and splits tetrahedra incident into v into two subset (see Figure 5.18 from right to left). The dependency relation is encoded in a Full-Edge Tree through a direct application of the view dependent tree [ESV99]. The encoding structure is similar to the one described in Section 5.3.1.

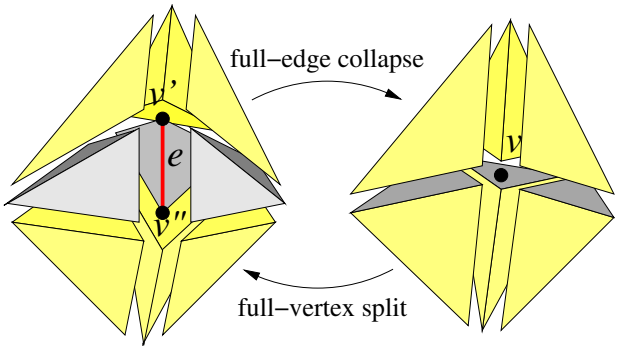


Figure 5.18: Full-edge collapse and vertex split on a tetrahedral mesh (exploded view). On the left, the set of tetrahedra incident into edge e is gray shaded. Such tetrahedra are collapsed to triangles. Yellow tetrahedra are affected by the edge collapse On the right, the new vertex v and the new tetrahedra incident in v are shown.

In an FET, only full-edge collapses at the midpoint of an edge are encoded thus the position and the field value of vertex v are obtained by linear interpolation from the corresponding information at v' and v".

The following information are encoded for each modification u:

- an offset vector, which is used to find the positions of vertices v' and v" from that of v and vice-versa;
- an offset value, which is used to obtain the field values at v' and v" from that of v and vice-versa;
- the error associated with the modification, with the maximum of the field error associated with the tetrahedra generated by full-edge collapse;
- a bit mask, which is used to partition the star of v.

The bit mask contains one bit for each tetrahedron incident at v. Incident tetrahedra are sorted lexicographically according to the triplets formed by their vertices different from v, where the vertices are sorted according to their index. Then, the following rule is applied: tetrahedra marked with 0 must replace v with v'; tetrahedra marked with 1 must replace v with v"; each triangular face shared by two differently marked tetrahedra must be expanded into a tetrahedron incident at both v' and v".

In order to resolve ambiguities in boundary configurations, the existence of dummy tetrahedra attached to the faces lying on the mesh boundary and incident to a dummy vertex is assumed. In the case of a boundary update u, the bit vector includes one bit for each dummy tetrahedron.

The cost of encoding the view dependent tree is equal to $12n$ Bytes (see Section 5.3.1). The encoding cost for modification can be evaluated as follows. The offsets of coordinate and field value are stored in 2 Bytes. Only edges such that the total number of adjacent vertices to their two endpoints is not larger than 32 are collapsed, and the number of tetrahedra incident in a vertex is bounded by 60. This means that the bit mask can be stored in 8 Bytes. The storage cost for the information associated with a single modification contributes for a cost of 18 Bytes, by using 2 Bytes for each error. The total cost of an FET data structure is $18n + 12n = 30n$ Bytes, plus the cost of the base mesh, which is negligible. Thus, it achieves a compression factor of about 3.09 or 5.81, compared with the mesh at full resolution encoded into an indexed data structure, or into an indexed data structure with adjacencies, respectively.

If an edge $e = (v', v")$ is collapsed to vertex v which is an internal point of e and not the midpoint of e, two offset vectors of coordinates and two field value offsets need to stored. Using these offsets coordinates and field values of v' and $v"$ are retrieved from ones of v while a full-vertex split is applied. In this case, the storage cost of an update is 26 Bytes, and the total cost of the FET data structure increases to $26n + 12n = 38n$ Bytes.

5.4.2 A Hierarchy of Tetrahedra

The Hierarchy of Tetrahedra (HT) is a multiresolution model based on a regular tetrahedral decomposition of the domain [LDFS01]. The refinement procedure is based on tetrahedron bisection.

The bisection rule for tetrahedra consists of replacing a tetrahedron σ with the two tetrahedra obtained by splitting σ through the middle point of its longest edge and by the plane passing through such point and the opposite edge in σ (see Figure 5.19a). The bisection rule is applied recursively to an initial decomposition of the cubic domain into six tetrahedra (see Figure 5.19b). Regardless of the number of times the tetrahedra are bisected, the resulting shapes always fall into one of three cases. These shapes are actually cyclic in that every three levels of decomposition result in a geometrically similar shape. Splitting a tetrahedron in the initial cube subdivision results in two tetrahedra with a shape identical to that obtained by splitting a pyramid with a square base in half along the diagonal of its base. We call such shape a *1/2 pyramid* (see Figure 5.19c). Splitting a 1/2 pyramid along its longest edge results in two tetrahedra whose shape we call a *1/4 pyramid* (see Figure 5.19d). Finally, splitting a 1/4 pyramid along its longest edge results in two tetrahedra whose shape we call a *1/8 pyramid* (see Figure 5.19e). Each of the six initial tetrahedra also has a 1/8 pyramid shape.

Tetrahedron bisection is a non-conforming modification, in the sense that it generates non-conforming meshes. To be able to extract conforming meshes, all tetrahedra sharing an

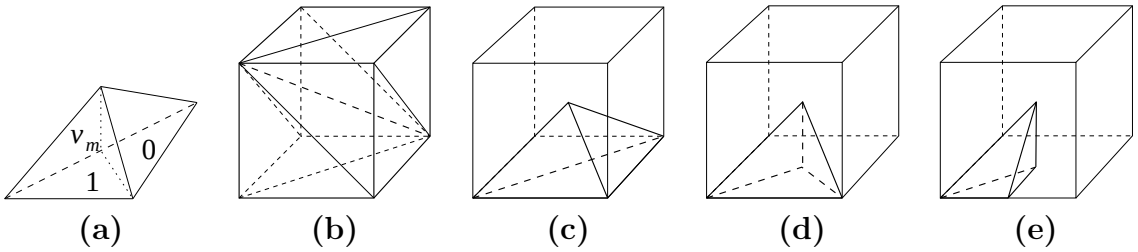


Figure 5.19: (a) An example of tetrahedron bisection. (b) Subdivision of the initial cubic domain into six tetrahedra. Examples of a 1/2 pyramid (c), of a 1/4 pyramid (d), and of a 1/8 pyramid (e).

edge must be split at the same time thus generating three kinds of tetrahedral clusters, called *diamonds*:

- an *axis aligned* diamond is formed by eight 1/2 pyramids, as only 1/2 pyramids share an axis-aligned edge;
- a *plane-aligned* diamond is formed by four 1/4 pyramids, as only 1/4 pyramids share a plane-aligned edge, i.e., on an edge parallel to one of the coordinate planes;
- a *non-aligned* diamond is formed by twelve 1/8 pyramids, as only 1/8 pyramids share an edge which is not aligned to a coordinate axis or plane.

An HT is encoded as a binary forest which describes the nested structure of the subdivision. It makes use of a linear representation for the hierarchy instead of a pointer-based data structure. The data structure consists of:

- a table containing the field values at the data points;
- a forest of six almost full binary trees, containing the errors associated with the tetrahedra encoded as an array. The trees describe the complete subdivision of the initial cube, except for the last level which corresponds to the tetrahedra of the reference mesh, that have a null error.

In a data set with n points, there are $6n$ tetrahedra in the reference mesh, and, thus, $6n$ internal tetrahedra. This yields a storage cost of $12n$ Bytes for the error values plus $2n$ Bytes for the field table, assuming two Bytes per error and field value, leading to a total cost of $14n$ Bytes.

Location codes, one for each tetrahedron σ, are used to index the field table. In each location code, the first number is the level of σ in the tree, and the second number describes the path

from the root of the tree to σ. The location code is defined on the basis of a labeling scheme for the children of a tetrahedron and for the vertices of these children in the hierarchy. The forest can be traversed using only local computations to determine where we are in space: the only computation is finding the midpoint of the longest edge. All other vertices can be obtained directly from the vertices of the parent tetrahedron. Diamonds are computed on the fly when extracting the mesh from an HT by using a worst case constant time neighbor finding algorithm [LDFS01].

5.5 Analysis and Comparisons

We have performed experiments on a number of regular, curvilinear and irregular data sets. Regular as well as curvilinear data sets are tetrahedralized by splitting each hexahedral cell into five tetrahedra. In our experiments, we have used the following data sets :

- *Buckminster Fullerene (Buckyball)* (Robert Haimes, MIT) represents a carbon molecule having 60 atoms arranged in the form of a soccer-ball, or truncated icosahedron, having 262,144 vertices and 1,250,235 tetrahedra.
- *Plasma* (Visual Computing Group, Italian National Research Council) is a synthetic data set representing Perlin noise, having 274,625 vertices and 1,310,720 tetrahedra.
- *Blunt Fin* (C.M. Hung and P.G. Buning, NASA) represents an airflow over a flat plate with a blunt fin rising from the plate. It has 40,948 vertices and 222,414 tetrahedra.
- *Liquid Oxygen Post* (S. E. Rogers, D. Kwak, U. Kaul, NASA) represents the flow of liquid oxygen across a flat plate with a cylindrical post rising perpendicular to the plate. It has 108,300 vertices and 616,050 tetrahedra.
- *Langley Fighter* (Neely and Batina, NASA) represents data for the Langley Fighter, from a wind tunnel model developed at NASA Langley Research Center. It has 13,832 vertices and 70,125 tetrahedra.
- *San Fernando* (D.R.O'Hallaron and J.R.Shewchuk, Quake project) represents the simulation of an earthquake in the San Fernando Valley in Southern California. The field value is associated to the p-wave velocity. It has 378,747 vertices and 2,067,739 tetrahedra.

Tables 5.1-5.6 report for each dataset the encoding cost of:

- the full resolution mesh with an indexed data structure (*Indexed*);

	Indexed	Ind+Adj	Explicit	VMT	HE-MT	HET	FET
Base mesh			79	79	79	79	79
DAG			1568	1167	1167	480	475
Modifications			10554	763	604	732	709
Total	4115	7750	12200	2008	1849	1291	1263
Bytes/mod.			307.6	49.0	44.9	30.8	30.1

Table 5.1: Space required for storing the Blunt Fin dataset with different data structures. All values are in KBytes. The last row reports the average size of a modification expressed in Bytes.

	Indexed	Ind+Adj	Explicit	VMT	HE-MT	HET	FET
Base mesh			206	206	206	206	206
DAG			468	349	349	151	150
Modifications			3468	238	186	225	215
Total	1312	2462	4142	793	741	581	570
Bytes/mod.			329.7	49.2	44.8	31.5	30.5

Table 5.2: Space required for storing the Fighter dataset with different data structures. All values are in KBytes. The last row reports the average size of a modification expressed in Bytes.

	Indexed	Ind+Adj	Explicit	VMT	HE-MT	HET	FET
Base mesh			1	1	1	1	1
DAG			10532	7857	7857	3252	3218
Modifications			69848	5157	4084	4956	4827
Total	24771	46324	80381	13014	11942	8208	8046
Bytes/mod.			299.7	48.5	44.5	30.6	30.0

Table 5.3: Space required for storing the Plasma dataset with different data structures. All values are in KBytes. The last row reports the average size of a modification expressed in Bytes.

	Indexed	Ind+Adj	Explicit	VMT	HE-MT	HET	FET
Base mesh			164	164	164	164	164
DAG			4539	3335	3335	1274	1261
Modifications			32601	2167	1662	2002	1886
Total	11318	21367	37305	5666	5161	3441	3311
Bytes/mod.			354.5	52.5	47.7	31.3	30.0

Table 5.4: Space required for storing the Oxygen Post dataset with different data structures. All values are in KBytes. The last row reports the average size of a modification expressed in Bytes.

	Indexed	Ind+Adj	Explicit	VMT	HE-MT	HET	FET
Base mesh			108	108	108	108	108
DAG			11659	8500	8500	3099	3067
Modifications			73192	5119	3973	4803	4597
Total	23631	44190	84960	13728	12582	8011	7773
Bytes/mod.			332.2	53.3	48.8	30.9	30.0

Table 5.5: Space required for storing the Buckyball dataset with different data structures. All values are in KBytes. The last row reports the average size of a modification expressed in Bytes.

	Indexed	Ind+Adj	Explicit	VMT	HE-MT	HET	FET
Base mesh			8	8	8	8	8
DAG			15345	11347	11347	4484	4438
Modifications			106379	7394	5758	6960	6656
Total	38226	72014	121732	18749	17114	11452	11103
Bytes/mod.			329.2	50.7	46.3	30.9	30.0

Table 5.6: Space required for storing the San Fernando dataset with different data structures. All values are in KBytes. The last row reports the average size of a modification expressed in Bytes.

	VMT	HE-MT	HET	FET
Indexed	1.89	2.06	3.01	3.09
Ind+Adj	3.56	3.88	5.65	5.81
Explicit	6.12	6.67	9.74	10.01
VMT	1	1.09	1.58	1.63
HE-MT	0.92	1	1.45	1.49
HET	0.63	0.69	1	1.03
FET	0.62	0.67	0.97	1

Table 5.7: Average compression ratio, higher values are better.

- the full resolution mesh with an extended indexed data structure (*Ind+Adj*) which, for each tetrahedron σ, encodes relations $R_{30}(\sigma)$, $R_{33}(\sigma)$ and $R^*_{03}(v)$. Relation $R^*_{30}(v)$ stores, for each vertex v, one of its incident tetrahedra.

We also report the storage cost of the following multi resolution data structure:

- the Explicit MT (*Explicit*);
- the Vertex-based MT (*VMT*);
- the Half-edge MT with dependency relation encoded as a DAG [KG98] (*HE-MT*);
- the Half-edge MT with dependency relation encoded with view-dependent tree (*HET*);
- and the Full-edge MT (*FET*).

We report the cost of encoding the base mesh, the dependency relation and modifications. Such values are in KBytes. For each multiresolution model we also report the average size of a modification, expressed in Bytes.

Table 5.7 compares the average encoding costs among compact data structures. It is easy to recognize that the vertex-based 3D MT introduces only a small overhead compared with the Half-Edge MT, but it offers higher flexibility because it can handle also general vertex insertions/removals. The HET has nearly the same storage cost of the FET. It is interesting to note that compact multiresolution data structures achieve a compression factor between 47% and 68% with respect to the indexed data structure describing the mesh at full resolution and a compression factor between 72% and 83% with respect to the extended indexed data structure with adjacencies representing the mesh at full resolution. If we compare to the Explicit MT, compression ratio grows up to $84 - 90\%$.

We have compared the performances of MTs built through half-edge collapse and full-edge collapse, and of the HT, on the following selective refinement queries:

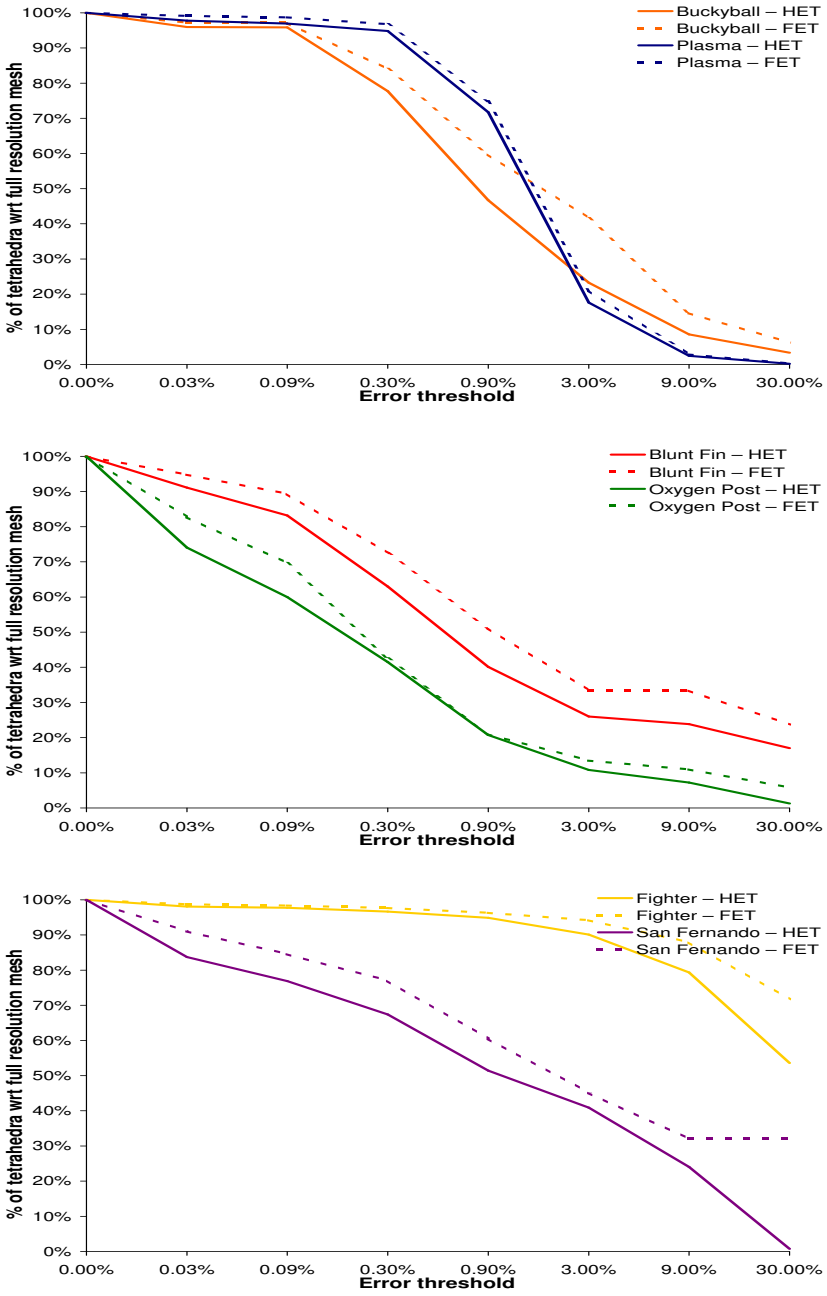


Figure 5.20: Uniform LOD query: blue and red lines represent the size of the extracted mesh with respect to the size of the full resolution mesh. Dashed lines represent multiresolution models built through full-edge collapse while continuous line are based on half-edge collapse. From top to bottom: regular (Buckyball and Plasma), curvilinear (Blunt Fin and Liquid Oxygen Post) and irregular (Langley Fighter and San Fernando) data sets.

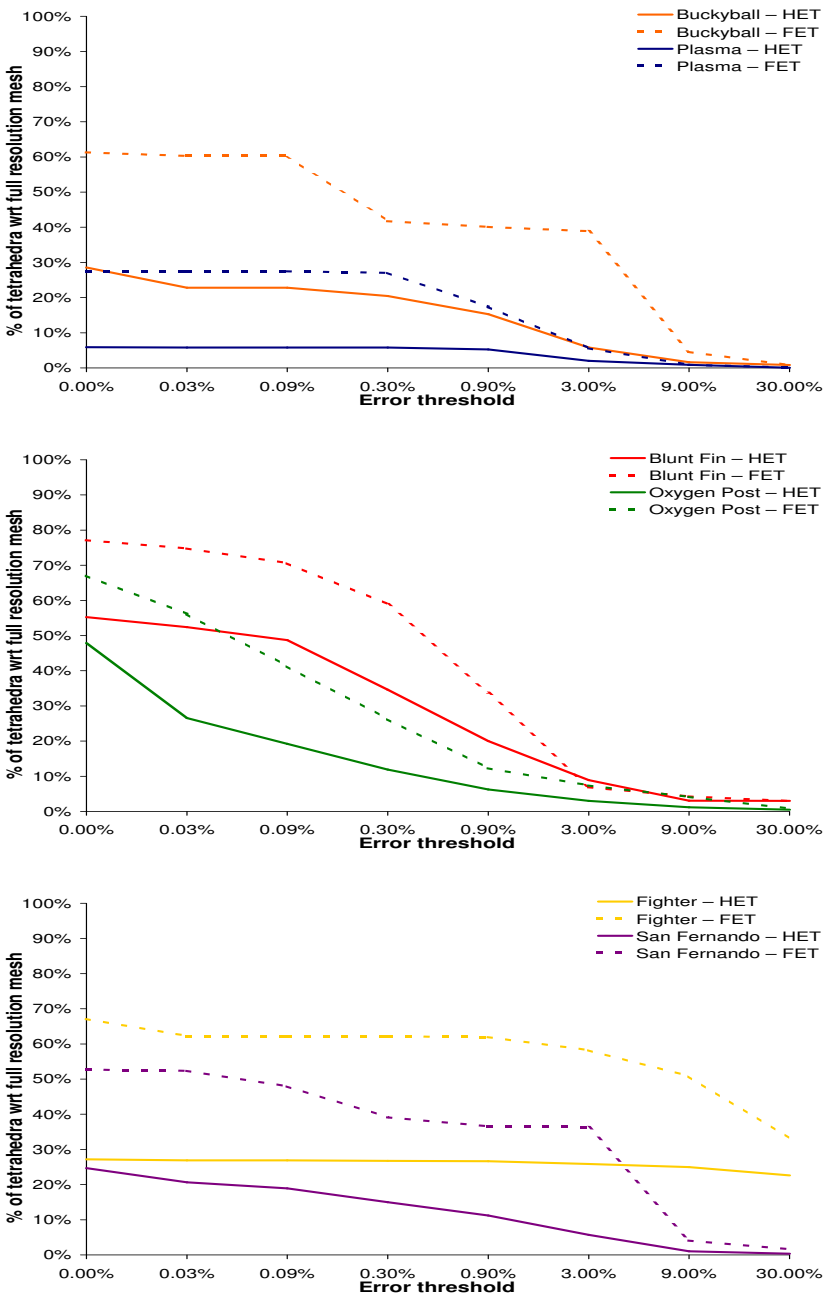


Figure 5.21: Variable LOD based on a region of interest. From top to bottom: regular (Buckyball and Plasma), curvilinear (Blunt Fin and Liquid Oxygen Post) and irregular (Langley Fighter and San Fernando) data sets.

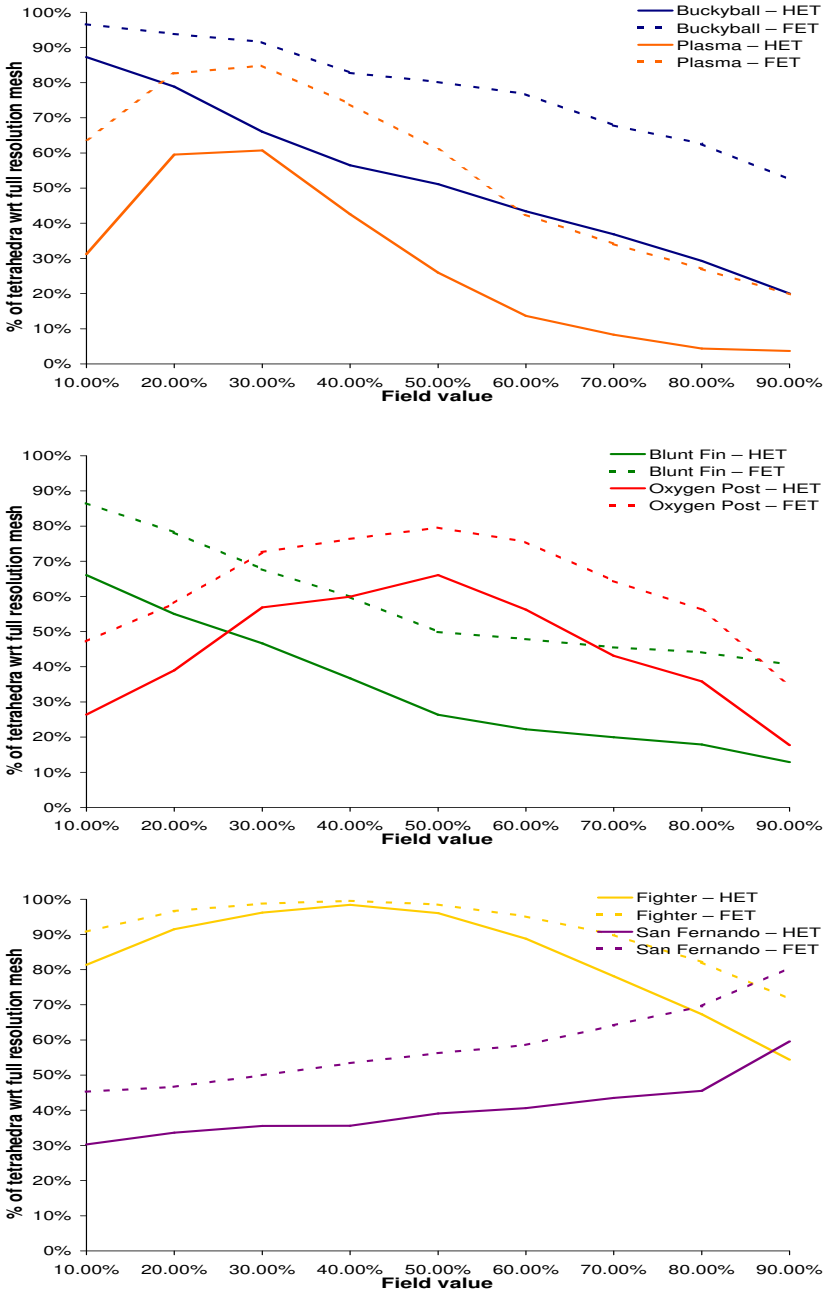


Figure 5.22: Variable LOD based on field values. From top to bottom: regular (Buckyball and Plasma), curvilinear (Blunt Fin and Liquid Oxygen Post) and irregular (Langley Fighter and San Fernando) data sets.

1. *Uniform LOD*: extraction at uniform LOD over the whole data set, with a given error threshold varying between zero (extracting the reference mesh) and the maximum error value (extracting the base mesh);

2. *Variable LOD in a box*: extraction of a mesh at a variable LOD with a an error below a given threshold inside the box and equal to the maximum error value outside it. We consider the box at a fixed position, and the threshold inside the box varying between zero and maximum error;

3. *Variable LOD based on a field value*: extraction of a mesh at a variable LOD with an iso-surface error below a certain threshold on the tetrahedra intersecting the specified iso-surface, and equal to the maximum error value elsewhere. The isovalue varies between the minimum and the maximum field value in the given data set.

Figures 5.20, 5.21 and 5.22 report numerical results on such experiments in the form of graphs. In order to improve readability, we have divided each experiment into three groups referring to extractions from the two regular data sets (Buckyball and Plasma), from the two curvilinear data sets (Blunt Fin and Liquid Oxygen Post), and from two irregular data sets (Langley Fighter and San Fernando), that we have used in our experiments. Figure 5.20 reports the number of tetrahedra in the meshes extracted at uniform LOD as a function of the error threshold. Figure 5.21 reports the number of tetrahedra in the meshes extracted at a variable LOD based on a box as a function of the error threshold. Figure 5.22 reports the number of tetrahedra in the meshes extracted at a variable LOD based on the field value as a function of the isovalue. For each experiment, we report the results obtained from both the half-edge and the full-edge building policy. On average, we see that the half-edge extracts meshes that, for a given error threshold, are 34% smaller than those extracted from the full-edge, and this gain can be up to about 75% for large error thresholds (coarse approximation).

We also compared the Half-Edge MT with the Hierarchy of Tetrahedra. We have performed such comparisons only on regular datasets since HT can handle only such kind of data [LDFS01].

The first comparison that is for a uniform LOD query. Figure 5.23 on top shows the number of tetrahedra in the extracted mesh for different error thresholds. From such graphs we see that the meshes extracted from an HT has more tetrahedra than those extracted from an MT. The main reason for this is that the MT is more flexible in choosing which tetrahedra to split (e.g., when applying a vertex split operation), while in the HT these are determined by the fixed recursive decomposition rule. Also, note that, when we perform a vertex split in the MT, we introduce 5 tetrahedra on average, and when we perform a (clustered) modification in the HT, we introduce 6 tetrahedra on average.

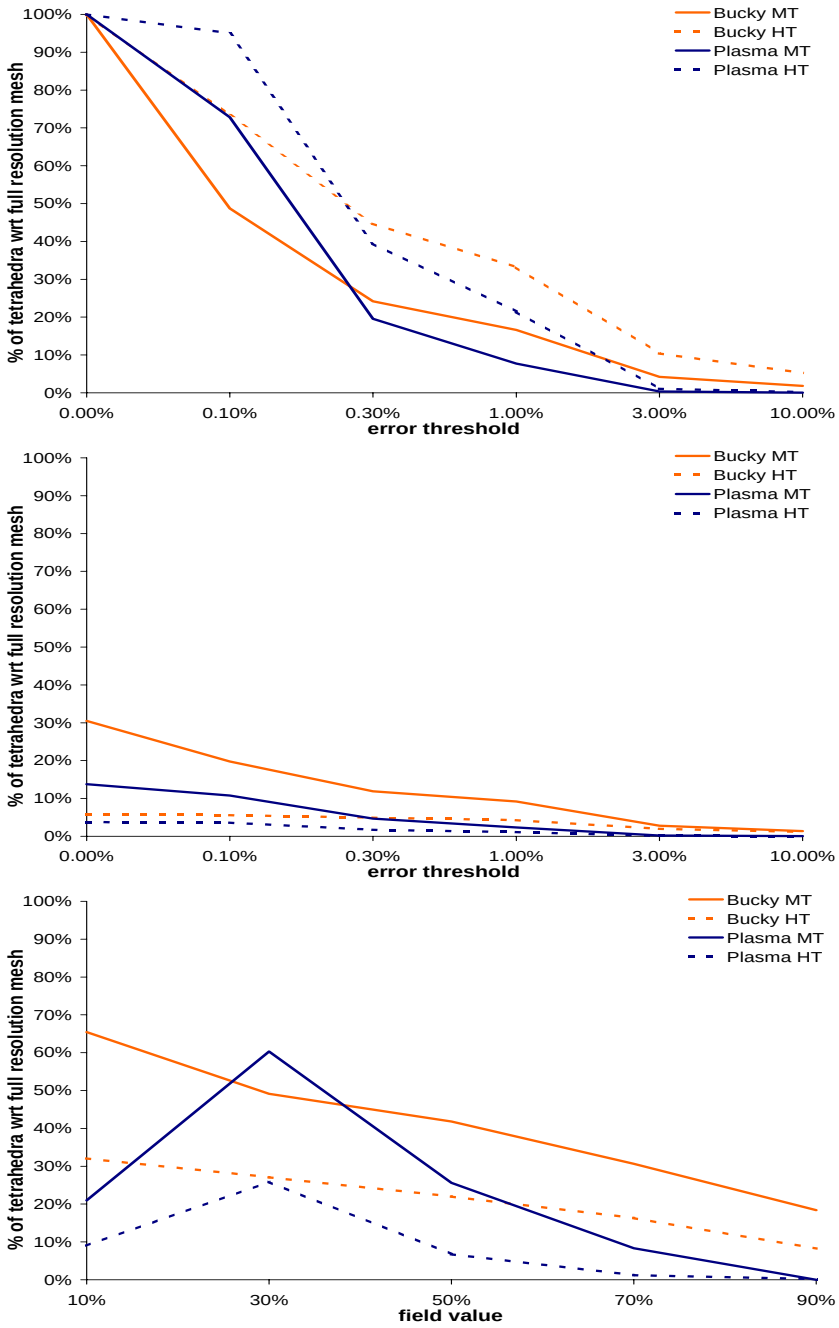
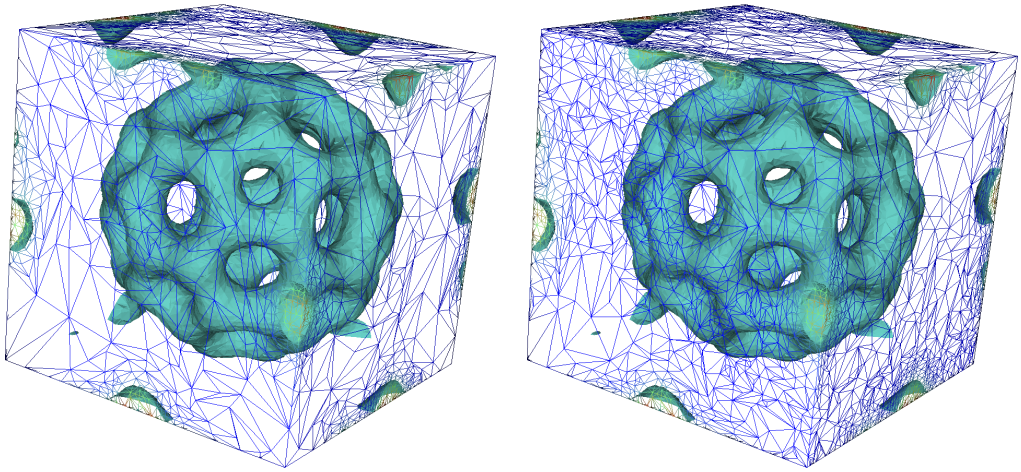


Figure 5.23: Comparison between Full-edge MT and HT multiresolution models performed on Buckyball and Plasma datasets. From top to bottom: query at uniform LOD, variable LOD based on a region of interest, and variable LOD based on a field value.

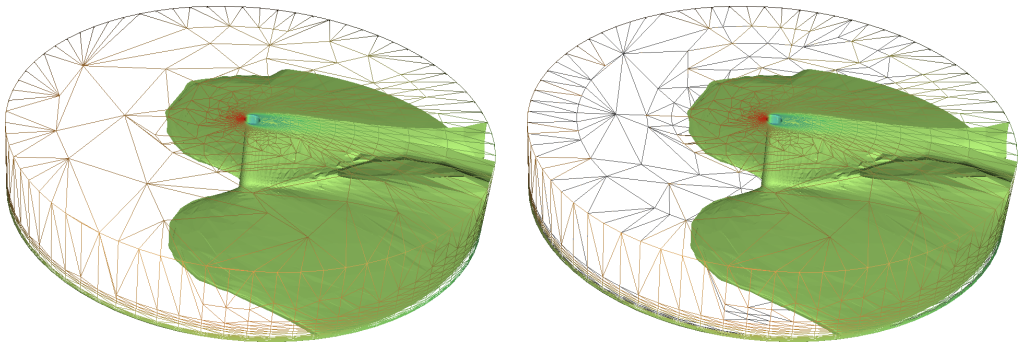


(a) **(b)**

Figure 5.24: Extraction at uniform resolution from Buckyball data set, the error threshold is equal 0.9% of the range of the field value. (a) Half-Edge MT - 122,916 vertices and 583,839 tetrahedra, (b) Full-Edge MT - 181,855 vertices and 853,009 tetrahedra.

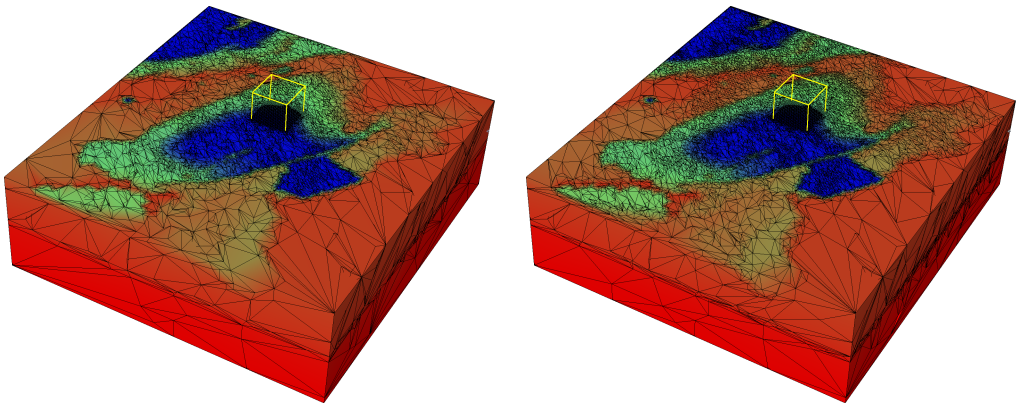
The second comparison deals with extracting a mesh at variable LOD in a region of interest. For our tests, we have chosen an axis-aligned box as region of interest. We use a number of different boxes at different positions and we consider the average number of tetrahedra in the resulting meshes, while varying the error threshold for the field values inside the boxes (see Figure 5.23 in the middle). Note that the number of tetrahedra in the HT is less than in the MT. This is due to the regular decomposition used in the HT, which means that the tetrahedra are aligned with the faces of the box (i.e., they are axis-parallel). In particular, we know that each of the tetrahedra in the HT has at least one face that is axis-parallel. Thus, as soon as we cross the boundary of a box, we can start having tetrahedra with larger faces, thereby having fewer tetrahedra than in the MT where the tetrahedra may be forced to intersect the boundary of the box, thereby delaying the merging process.

The third comparison that is for a query that extracts a mesh at variable LOD based on the field value (i.e., on the isosurface). A given error threshold is allowed in the tetrahedra intersected by the isosurface while a larger error threshold is allowed elsewhere. Figures 5.23 at the bottom show the resulting number of tetrahedra in the extracted mesh for the Smallbucky and Plasma data sets. As in the box query, the number of tetrahedra in the mesh is less for the HT than the MT. Thus, the HT is better than the MT at pruning the irrelevant tetrahedra from the isosurface field value that defines the query. Of course, it could also be argued that in the MT, the same error value is associated with all of the tetrahedra in the conforming update, while in the HT, an error value is associated with each tetrahedron. Thus, we could obtain better performance with the MT if we stored an error value with each tetrahedron, but this would increase the storage cost.



(a) **(b)**

Figure 5.25: Extraction at variable resolution from Liquid Oxygen Post data set with focus on an isovalue, the error threshold is 0.09% at such isovalue, arbitrary large elsewhere. (a) Half-Edge MT - 34,909 vertices and 195,449 tetrahedra, (b) Full-Edge MT - 50,458 vertices and 282,782 tetrahedra.



(a) **(b)**

f

Figure 5.26: Extraction at variable resolution from San Fernando data set with focus in a box, the error inside the box is null, arbitrary large elsewhere. (a) Half-Edge MT - 94,811 vertices and 509,964 tetrahedra, (b) Full-Edge MT - 202,674 vertices and 1,090,135 tetrahedra.

Figure 5.24 (a) and (b) show two meshes extracted at uniform LOD from an HET and from an FET, respectively, generated from the Buckyball data set. The error threshold over the whole domain is equal to 0.9% of the absolute value of the difference between the maximum and the minimum field value. The mesh extracted from the HET contains 122,916 vertices and 583,839 tetrahedra, while the one extracted from the FET contains 181,855 vertices and 853,009 tetrahedra.

Figures 5.25 (a) and (b) show two meshes extracted at a variable LOD based on field value from an HET and from an FET, respectively, generated from the Oxygen Post data set. We have selected 2.42 as field value, and an error threshold equal to 0.09% along the isosurfaces of such field value, while the error can be arbitrarily large outside. The mesh extracted from the HET contains 34,909 vertices and 195,449 tetrahedra, while the one extracted from the FET contains 50,458 vertices and 282,782 tetrahedra.

Finally, Figures 5.26 (a) and (b) show two meshes extracted at a variable LOD with focus on a box from an HET and from an FET, respectively, generated from the San Fernando data set. The error inside the box is equal to zero, and arbitrarily large outside. The mesh extracted from the HET contains 94,811 vertices and 509,964 tetrahedra, while the one extracted from the FET contains 202,674 vertices and 1,090,135 tetrahedra. These results clearly show that, by using and HET, we can achieve the same quality in accuracy with a smaller number of tetrahedra.

Our current implementation has been developed in C++. Tests are based on a P IV 2.6 Ghz workstation with GNU/Linux. We have found that, on average, the top-down selective refinement algorithm can extract from an HET about 180K tetrahedra per second. We have also experimented on an implementation of the same algorithm on an explicit data structure for encoding the LOD model, in which all tetrahedra are described explicitly and the dependency relation is encoded as a DAG [DFMPS02]. On this explicit data structure, the top-down algorithm can extract, on average, about 330K tetrahedra per second. On the other hand, the explicit data structure needs 2.2 times the space used to encode the mesh at full resolution as an extended indexed data structure with adjacencies, while the HET requires about 3/4 of the space used by the mesh at full resolution.

Chapter 6

An out-of-core approach to multiresolution modeling

The size of available datasets is becoming larger and larger and even high-end workstations cannot handle such huge datasets. In-core data structures for multiresolution modeling help handling and visualizing large datasets, but some large datasets cannot fit in main memory even on standard graphic workstations. If we consider a workstation with 2 Gigabytes of RAM, about 100 Megabytes are reserved for the operative system, and, thus, the remaining space has to be subdivided among the multiresolution data structure, the auxiliary data structures used to perform selective refinement, and the data structures encoding the extracted mesh. The extracted mesh is usually much smaller than the full resolution mesh. We assume that encoding the extracted mesh requires half of the space required by storing the reference mesh. Using our implementation of the explicit MT, described in Section 4.4, we can fit and query in main memory a 2D multiresolution model built from a full resolution mesh of the order of million triangles. In case of 3D models the size decreases to 16 millions of tetrahedra. If we adopt a compact multiresolution data structure, as those described in Sections 4.5, 5.1, 5.2, 5.3 and 5.4.1, the size of the data structures is much smaller, but we need to encode the extracted mesh with adjacencies. In case of 3D models, we can handle at most a multiresolution model built from a reference mesh containing as much as 65 millions of tetrahedra. The only chance to handle larger datasets is to store them in secondary storage and develop external-memory algorithms for building and querying a multiresolution model.

For this reason, we investigate how to encode our multiresolution data structures in secondary storage. Our goal is to define an out-of-core multiresolution data structure that can store and perform queries on a large static dataset on a consumer hardware. Static means that we are not interested in updating the multiresolution model. One of the desirable properties of an out-of-core data structure is that it scales well, that is, if the data struc-

ture can fit in main memory, the overhead should be low. We should be able to perform selective refinement queries on arbitrary large models, with a (possibly) minimum number of page faults for each query.

6.1 Background

6.1.1 Memory Hierarchy

Most computer systems usually contain a hierarchy of memory levels, where each level has its own cost and performance characteristics. At the highest level, CPU registers and caches are built with the fastest and most expensive memory. Main memory is usually based on dynamic random access memory (DRAM), or on its evolutions. Lower-level memory is based on inexpensive, but slower, magnetic disks, which are used for external mass storage.

Most modern programming languages are based upon a programming model in which memory consists of a single uniform address space. The notion of virtual memory allows the address space to be larger than what can fit in the main memory of the computer. Most programs assume that all memory references can be accessed with the same access time. In many cases, such an assumption is reasonable, especially when the data sets are not large. However, not all memory references are equal and accessing the data in the highest levels of memory is orders of magnitude faster than accessing the data at the lower levels. For example, loading a register takes on the order of a nanosecond (10^9 seconds), and accessing internal memory takes tens of nanoseconds, but the latency of accessing data from a disk is several milliseconds (10^3 seconds), which is about one million times slower.

In applications that process massive amounts of data, the Input/Output communication (or simply I/O) between levels of memory is often the bottleneck. Many computer programs are characterized by locality in memory references: certain data are referenced repeatedly for a while, and then the program shifts to other sets of data.

Modern operating systems take advantage of such locality by introducing the notion of *working set*: it roughly corresponds to the recently referenced data items. If the working set is small, it can be cached in high-speed memory so that access to it is fast. Caching and prefetching heuristics have been developed to reduce the number of occurrences of a fault, in which the referenced data item is not in the cache and must be retrieved by an I/O from a lower level of memory. For example, in a page fault, an I/O is needed to retrieve a disk page from disk and bring it into internal memory. Caching and prefetching methods are typically designed to be general-purpose and, thus, they cannot be expected to take full advantage of the locality present in every computation.

Some computations are inherently non-local, and even with perfect cache management, they perform large amounts of I/Os, and thus, they perform badly. This can often happen with large datasets. It is possible to obtain large performance improvements by incorporating locality directly into the algorithm design and by explicitly managing the contents of each level of the memory hierarchy, bypassing the virtual memory system.

In the case of the queries on a large multiresolution model, each atomic operation is local, since it operates on a small subset of the whole domain, but each modification is performed only once, and automatic definition of a working set is nearly useless. Moreover, selective refinement algorithms visit and perform modifications in a dynamically selected order, so it is impossible to forecast the best storing order. We will address this problem by grouping sets of modifications, based on spatial coherence, or on the dependency relation, thus minimizing the number of I/O operations.

We have designed out-of-core algorithms and data structures that explicitly manage data placement and movement between external and internal memory. We concentrate our studies on the I/O communication between the random access internal memory and the magnetic disk external memory, where the relative difference in access speeds is most evident. We therefore use the term I/O to designate the communication between the internal memory and the disks.

6.1.2 Computational Model

In order to describe our approach, we adopt the parallel disk model (PDM) described by Vitter and Shriver in [VS94], that we briefly summarize here. The PDM provides an elegant and reasonably accurate model for analyzing the relative performance of Out-Of-Core (OOC) algorithms and data structures. The three main performance measures of the PDM are the number of I/O operations, the disk space usage, and the CPU time. In geometric modeling, CPU time is mainly spent in floating point computation and is nearly unaffected by the out-of-core design, thus we focus on the first two measures.

Out-of-core algorithms explicitly control data loading and storing. Thus, it is important to have a simple, but reasonably accurate, model of the memory system and of its characteristics. Magnetic disks consist of one or more rotating platters and one read/write head per platter surface. The data are stored on the platters in concentric circles, called *tracks*. To read or write a data item at a certain address on disk, the read/write head must mechanically seek the correct track and then wait for the desired address. The seek time to move from one track to another is often of the order of 3 to 10 milliseconds, and the average rotational latency, which is the time for half a revolution, has the same order of magnitude. In order to amortize this delay, it is better to transfer a large contiguous group of data items, called a *block*. Similar considerations apply to all levels in the memory

hierarchy.

Even if an application can optimize memory accesses to exploit locality and take advantage of disk block transfer, there is still a large gap between access performances of internal and external memory. This gap is growing, since the latency and bandwidth of memory chips are improving more quickly than those of disks. Parallel processors further enlarge the gap. Fast storage systems, such as RAID (Redundant Array of Inexpensive Disks) systems, deploy multiple disks in order to get additional bandwidth.

The following parameters are used for capturing the main properties of magnetic disks and multiple disk systems:

- N = problem size (in units of data items);
- M = internal memory size (in units of data items);
- B = block transfer size (in units of data items);
- D = number of independent disk drives;
- P = number of CPUs.

Note that $M < N$, and $1 < D \cdot B < M/2$. We simplify the PDM model with the assumption that we are working on a single-processor workstation ($P = 1$). The data items are assumed to be of fixed length. In a single I/O, each of the D disks can simultaneously transfer a block of B contiguous data items.

Vitter [VS94] also defined two more performance parameters, namely Q = number of input queries (for a batched problem), and Z = query output size (in units of data items). Since we are not interested into batched problems, we consider only the output size Z. PDM also assumes that the input data are initially striped across the D disks, in units of blocks, and output data are similarly striped. Striped format allows a file of N data items to be read or written in $O(N/DB)$ I/Os, which is optimal.

Practical Modeling Considerations The track size is a fixed parameter, related to disk hardware. For most disks it is between 50 and 200 KBytes. Basically, the track size for any given disk depends on the radius of the track. Sets of adjacent tracks are usually formatted to have the same track size. Thus, there are typically only a small number of different track sizes for a given disk. Outer tracks can be up to 50% larger than inner tracks.

The minimum block transfer size imposed by hardware is often 512 Bytes, but operating systems generally use a larger block size, such as 8 KBytes. It is possible to use larger

logical blocks and further reduce the relative significance of seek and rotational latency, but the total time for each I/O will increase accordingly. Once the block transfer size becomes larger than the track size, the total time per I/O grows linearly with the block size.

Disk Striping for Multiple Disks It is much simpler to develop an algorithm for the single-disk case, where $D = 1$, than for the multiple-disk case ($D > 1$). In a workstation with D disks, we can consider them like a large logical drive, and store data uniformly spanned on the D disks. The basic unit, called *stripe*, is composed of a block in each disk. Disk striping is a paradigm that simplify the programming task with multiple disks: I/Os are permitted only on entire stripes, one stripe at a time. The effect of striping is that the D disks behave as a single logical disk, but with a larger logical block: its size becomes $D \cdot B$. We can thus apply the paradigm of disk striping to automatically convert an algorithm designed to use a single disk with block size DB into an algorithm for use on D disks each with block size B: in the single-disk algorithm, each I/O step transmits one block of size DB; in the D-disk algorithm, each I/O step consists of D simultaneous block transfers of size B each. The number of I/O steps in both algorithms is the same; in each I/O step, the DB items transferred by the two algorithms are identical. This allow us to reduce the analysis of our out-of-core algorithm to the case $D = 1$.

6.2 I/O Operations in Selective Refinement

The selective refinement algorithms, described in Section 4.3, perform a traversal of a subset of the modifications of a Multi-Tessellation $\mathcal{M}$. The visit is guided by the dependency relation associated with $\mathcal{M}$, and by the selection criterion. In this Section, we consider a top-down, depth-first algorithm, but similar considerations can be applied to any selective refinement algorithm. We analyze every piece of pseudo-code describing the top-down selective-refinement algorithm and focus on I/O operations, and geometric, or integer, computations. We report the pseudo-code here for completeness:

```
Procedure SELECTIVE_REFINEMENT(M, τ)
begin
/* create an initial empty set of modifications and empty mesh */
   set S = empty_set()
   mesh Σ_S = empty_mesh()
/* start the recursive visit from the base modification */
   TOPDOWN_DEPTHFIRST_REFINEMENT(M, τ, S, Σ_S, BASE(M))
end SELECTIVE_REFINEMENT
```

Procedure SELECTIVE_REFINEMENT algorithm initialize the set of modification S and the current mesh Σ_S, then it simply performs an I/O operation, loading the base mesh

BASE($\mathcal{M}$), and then calls procedure TOPDOWN_DEPTHFIRST_REFINEMENT. Other operations just initialize the set S of current modifications and the mesh Σ_S associated with set S as the empty mesh.

```
Procedure TOPDOWN_DEPTHFIRST_REFINEMENT(M, τ, S, Σ_S, u)
begin
/* add the modification and its missing ancestors */
   FORCE_REFINE(M, S, Σ_S, u)
   for each u' in CHILDREN(M, u) do
      if not IS_ACCEPTABLE(u', τ) then
/* start recursion on required children */
         TOPDOWN_DEPTHFIRST_REFINEMENT(M, τ, S, Σ_S, u')
end TOPDOWN_DEPTHFIRST_REFINEMENT
```

Procedure TOPDOWN_DEPTHFIRST_REFINEMENT calls two primitives: CHILDREN and IS_ACCEPTABLE. Given a modification u, the first primitive returns the set of modifications which depend on u. When storing the MT with the explicit data structure (see Section 4.4), this can be done without I/O operations, since each modification stores pointers to its parents and to its children. By using compact data structures, it could require several I/O operations. In the worst case, the DAG encoding proposed by Klein and Gumhold [KG98] could require a number of I/O operations which is quadratic in the number of children of a given modification.

The cost of primitive IS_ACCEPTABLE on a modification u strongly depends on the filter selected by the user for the query:

- in case of a query at a uniform resolution, IS_ACCEPTABLE simply checks the error threshold associated with modification u. Usually, it requires one I/O to load the modification and a simple floating-point test;
- in case of queries based on a field value, IS_ACCEPTABLE checks the error threshold associated to the modification $u = (u^-, u^+)$ and the minimum and maximum field value of each simplex in u^-. Usually, it requires an I/O operation to load the modification and some integer operations to decode/test simplices in u^-;
- a query based on a region of interest requires decoding of the simplices in u^-, computing the intersection between the region of interest and simplices in u^-, taking into account the approximation error. It usually requires an I/O operation to load the modification, and several integer and floating-point computations.

In case of uniform resolution queries, it is possible to reduce the number of I/O operations by associating the error threshold of a modification u' with each pointer representing an arc

(u, u') of the DAG. It is also possible to store the minimum and maximum field value, or the bounding box of each modification. This allows us to load only modifications required by the specific LOD criterion, and minimize the number of I/O operations. Of course, this introduces a trade-off between storage space and running times.

```
Procedure FORCE_REFINE(M,S,Σ_S,u)
/* Recursively applies all ancestors of update u that are not in S. */
begin
   P = PARENTS(M, S, Σ_S, u)
   for each u' in P do
      FORCE_REFINE (M,S,Σ_S,u');
   endfor
   add_item(S, u)
   REFINE(M, S, Σ_S, u)
end FORCE_REFINE
```

Procedure FORCE_REFINE makes use of primitives PARENTS and REFINE. Moreover, it loads every modification returned by primitive PARENTS and recursively performs FORCE_REFINE on such modifications.

Efficiency of primitive REFINE is mainly related to the way modifications are encoded. Explicit encoding as in the explicit MT (see Subsection 4.4) results in a really fast execution, while compact encoding is quite time consuming. We assume that Σ_S is stored in main memory. This assumption is reasonable since the extracted mesh is usually only a small fraction (less than 20%) of the reference mesh.

Given a modification u, primitive PARENTS returns the set of modification from which u directly depends. Primitive PARENTS is used to retrieve ancestors of u that need to be added to the current set of modifications S. The number of missing ancestors can be arbitrary large.

Efficiency of primitive PARENTS in term of I/O operations is related to the data structure representing the dependency relation:

- with an explicit encoding of the dependency relation, the primitive PARENTS can return the parents of u without I/O operations;
- the compact encoding described by Klein and Gumhold [KG98] requires an I/O operation for each parent of u. It can introduce an overhead, with respect to the explicit DAG, since parents of u already present in S are not loaded in the explicit MT;
- when dependency relation encoded with the view-dependent tree, primitive ANCESTORS performs several computations on the current mesh in order to retrieve (rele-

vant) neighbors. This does not requires I/O. Only vertices that need to be splitted are loaded in main memory.

We have analyzed just the occurrence of I/O operations based on a data structure designed for in-core computations. Each read operation loads only a few Bytes (usually less than one KByte), and this results in a tremendous waste of time due to disk seek time and latency (see Section 6.1). In order to reduce the number of I/O operations, we need to group modifications into small clusters, and store each of them in a disk block (or stripe). This can reduce seek time and rotational latency significantly.

Another improvement can be achieved by introducing a cluster cache, which is a portion of main memory storing a subset of the clusters already loaded. When the selective refinement algorithm tries to load a modification, the out-of-core engine looks for it in the cache. If it is missing, the corresponding cluster is loaded and stored in the cache for further uses. If the cache is full, we need to adopt a cache replacement policy like the *least recently used* or the *oldest*.

In a multi-thread environment, it is possible to implement the I/O process in a tread, and the selective refinement in another one. This can be used to implement a prefetching techniques. Clusters that are likely to be required by the selective refinement algorithm can be pre-loaded by the I/O tread, with a reduction of the total execution time. Prefetching does not reduce the number of I/O operations performed. Since we focus on minimizing the number of I/O operations, we have not considered up to now prefetching.

6.3 Clustering Techniques

In order to analyze and compare different clustering and caching techniques, we have developed a prototype of a multiresolution engine that organizes modifications in clusters, according to different clustering policies, estimates performances of selective refinement and counts the number of I/O operations. The aim is to be able to compare several clustering techniques in both a 2D and 3D scenarios. This prototype is built on top of the MT-library [Mag00].

Developing a good clustering policy requires some knowledge about the shape of the multiresolution model and its behavior with standard queries. Section 6.3.1 presents some analysis performed on the DAG representing the dependency relation.

Since our goal is to minimize query time, and most of that time is spent in loading data from disk, we need to minimize the number of I/O operations. We can achieve this result by grouping modifications in a convenient way for standard queries. The most common queries have been described in Section 4.2.3. We can also consider point location queries in

which focus on a single point. We can associate an error threshold to that point. Another useful query is the view-dependent query. In this case, the region of interest is a cone. Error threshold inside the cone is smoothly decreasing from high to medium resolution, according to the distance from an observer, while the error threshold is set to infinity elsewhere.

Some queries, like point location or some queries based on region of interest, focus on a subset of the domain, usually at a high resolution. Thus, we are interested in space partitioning techniques. On the contrary, uniform resolution queries are performed on the whole domain. Thus, they suggest a subdivision of the MT in layers. Each layer should guarantee the reduction of the approximation error. View-dependent queries are still queries based on a region of interest, but in this case the error smoothly decreases according to the distance from the observer. View-dependent queries combine somehow space partitioning and error-driven queries. Queries based on the field value are strongly dependent on the size and on the shape of the iso-contour. They can behave like point-location queries for really small isosurface, on the contrary they can be compared to uniform queries for large isosurface that span over the whole domain.

Another important constraint is due to the dependency relation. We cannot apply a modification without applying the modifications associated with its ancestors. So, we need to fulfill three different requirements at the same time: space partitioning, coherence in the dependency relation and subdivision according to the approximation error. Section 6.3.2 presents some clustering techniques based on traversal of the dependency relation or on properties of modifications, while Section 6.3.3 focuses on clustering techniques based on space partitioning.

6.3.1 Shape of the multi-resolution model

As seen in the previous chapters, the dependency relation defining a multi-resolution model can be described by a DAG in which the nodes represent modifications, and the arcs the direct dependencies. The DAG describing a large multiresolution model can have several millions of nodes and an even larger number of arcs. In a 3D MT built through a half-edge collapse the number of arcs is on average 5 times the number of nodes. We define the following quantities as measures of the "shape" of a multiresolution model $\mathcal{M}$. In $\mathcal{M}$, we denote as $u = (u^-, u^+)$ a modification and by R the modification which encodes the base mesh (R is the root of the DAG).

- in-degree $in(u)$: number of modifications from which u directly depends;
- out-degree $out(u)$: number of modifications directly depending from u;
- layer $l(u)$: length of the shortest path from the root R to u;

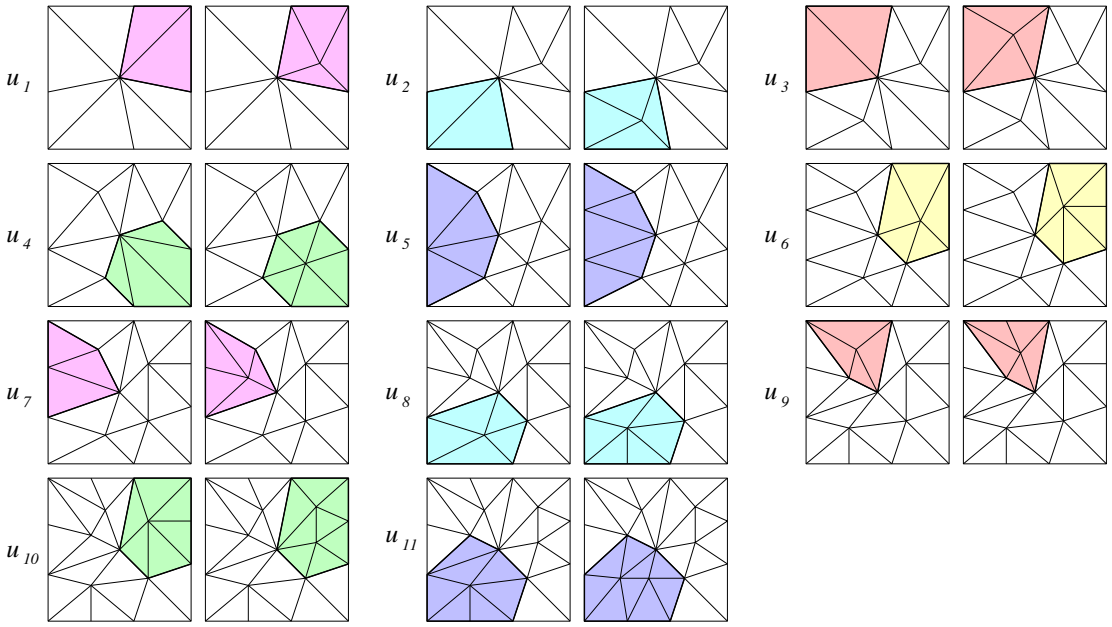


Figure 6.1: A sequence of modifications $u_i, i = 1, 2, \ldots 11$. For each modification u_i we show through colors on the left u_i^- and on the right u_i^+.

- level $L(u)$: length of the longest path from the root R to u;
- distance $d(u)$: defined recursively as follows: the distance of the root is zero $d(R) = 0$, for each generic modification u, $d(u) = \frac{\sum_{i=1}^{n} d(u_i)}{n} + 1$ where $u_1, \ldots u_n$ are the parents of u in $\mathcal{M}$.

Let us consider a simple 2D MT $\mathcal{M}$. Every technique presented in this chapter will be described referring to this toy example, while results are computed on real datasets. Figure 6.1 shows the sequence of modifications associated with this model. For each modification u_i, the left picture represents u_i^-, while the right one represents u_i^+. Figure 6.2 shows the dependency relation among the modifications depicted in Figure 6.1, represented as a DAG. Table 6.1 shows the shape measures computed on the MT $\mathcal{M}$ presented in Figure 6.2. Modifications are listed in the order generated by the simplification tool.

Tests performed on 3D volume data sets (Buckyball, Fighter, Plasma and San Fernando, for details see Section 5.5) show that, on average, a modification has an in-degree between one to seven, and an out-degree between two to nine (see Figure 6.3). This suggests that a tree-based partitioning is not adequate for an MT. Note that a multiresolution model, describing a nested tetrahedral hierarchy based on tetrahedral bisection (the hierarchy of tetrahedra described in section 5.4.2) is characterized by an average in-degree equals to 3 and an average out-degree equals to 6.

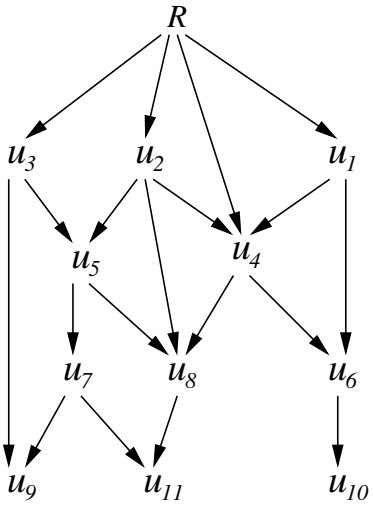


Figure 6.2: DAG representing the dependency relation among modifications depicted in Figure 6.1. Root R and drain D are added to the DAG

u	$in(u)$	$out(u)$	$l(u)$	$d(u)$	$L(u)$
R	0	4	0	0	0
u_3	1	2	1	1	1
u_2	1	3	1	1	1
u_1	1	2	1	1	1
u_5	2	2	2	2	2
u_4	3	2	1	$1.\overline{6}$	2
u_7	1	2	2	$2.\overline{5}$	3
u_8	3	1	2	3	3
u_6	2	1	2	$2.\overline{3}$	3
u_9	2	0	2	3	4
u_{11}	2	0	3	$3.\overline{7}$	4
u_{10}	1	0	3	$3.\overline{3}$	4

Table 6.1: Some of the shape measures associated to the modifications u_i: in-degree $in(u)$, out-degree $out(u)$, layer $l(u)$, distance from the root $d(u)$ and level $L(u)$.

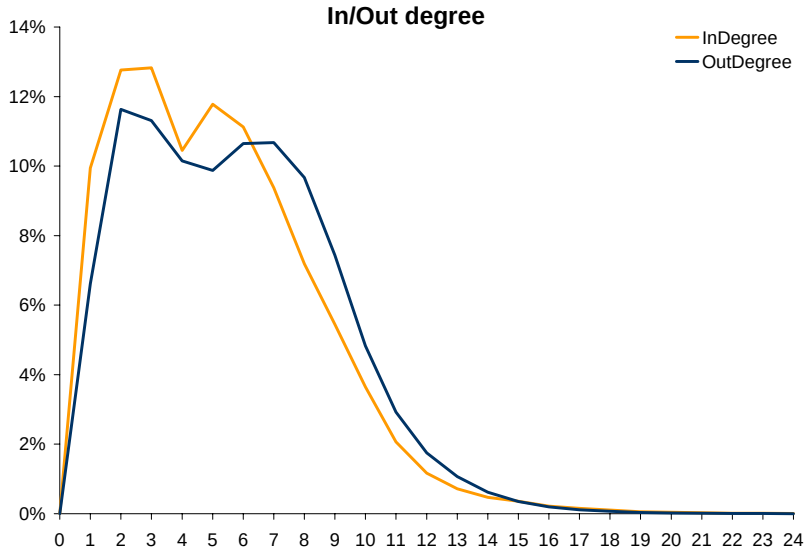


Figure 6.3: Average in-degree and out-degree of modifications in four 3D multiresolution models (built on Buckyball, Plasma, Fighter and San Fernando dataset)

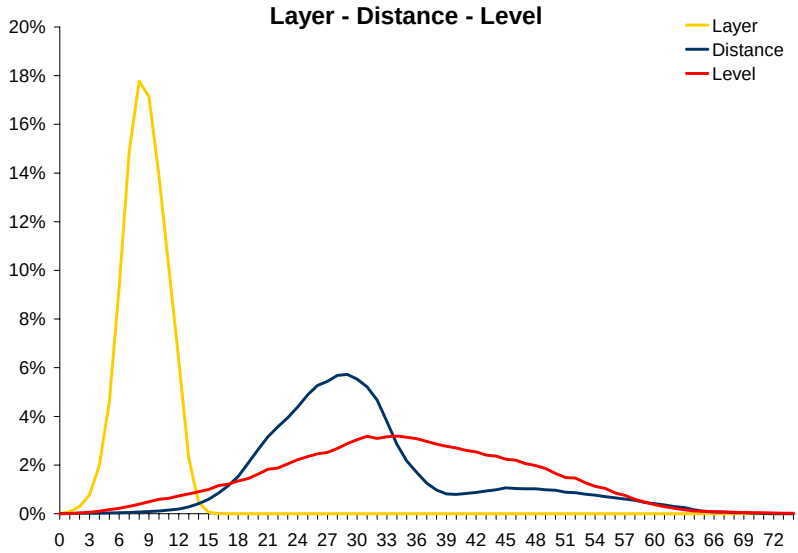


Figure 6.4: Average level, layer and depth of modifications in four 3D multiresolution model

Figure 6.4 shows the behavior of the other three measures on the same real datasets. As we can expect, the layer $l(u)$ is a spike, and most models are characterized by small maximum layer. Level $L(u)$ smoothly increases and then decreases, while distance $d(u)$ shows a quadratic increase in the first twenty layers, then it smoothly keeps increasing up to layer 30 and then starts a quite fast descent. The tail between layers 40 and 60 is due to the fact that we are analyzing datasets with different sizes.

6.3.2 Sorting of Modifications in an MT

In this Section, we describe clustering techniques based on linear sorting. For this purpose, we have defined several rules that can be used to sort set of modifications belonging to an MT:

- *random* (`Rnd`): it is implemented just for comparison purposes and is a completely random sorting of the modifications;
- *sequential* (`Seq`): modifications are kept in the same sequence generated by a top-down simplification procedure, or in reversed order in case of bottom-up simplification strategies. Usually the simplification process is error-driven, but there is no guarantee that the approximation error associated with modifications decreases monotonically;
- *approximation error* (`Err`): it sorts modifications according to their approximation error $e(u)$. It can take into account approximation error (in case of free form surfaces), field or isosurface error (for scalar fields);
- *depth-first* (`DFS`): it sorts modifications according to a depth-first traversal of the DAG. For modifications with more than one child, children are visited in the same order as they are returned by primitive CHILDREN;
- *breadth-first* (`BFS`): it sorts modifications according to a breadth-first traversal of the DAG. As in the `DFS` case, in case of modifications with more than one child, children are visited in the same order they are returned by primitive CHILDREN;
- *layer* (`Lyr`): modifications are sorted according to the value of their layer $l(u)$, by using a standard sorting algorithm;
- *level* (`Lev`): modifications are sorted according to the value of their level $L(u)$, by using a standard sorting algorithm;
- *distance* (`Ly2`): modifications are sorted according to the value of their distance, by using a standard sorting algorithm;

- *graph visit - breadth first* (`GrB`): it is similar to a breadth-first traversal, but before adding a modification u it checks if its ancestors are in the current set S of modifications. If they are not in S, they will be added by making procedure FORCE_REFINE. Modifications are added in the same order in which they are visited by procedure FORCE_REFINE. It simulates a breadth-first selective refinement algorithm for a query at uniform resolution with error threshold equal to 0;
- *graph visit - depth first* (`GrD`): it is a variation of `GrB`, based on a depth-first visit.

Note that `Lyr` and `BFS` give similar results. The only difference is that the modifications having the same value $l(u)$ are considered in arbitrary order by `Lyr`, while they are considered in the order in which they appear in the DAG encoding structure in `BFS`.

The sorted sequences we have obtained for the DAG in Figure 6.2 with the different methods are listed below:

`Seq`	$R, u_3, u_2, u_1, u_5,$	$u_4, u_7, u_8, u_6, u_9,$	u_{11}, u_{10}
`Err`	$R, u_1, u_2, u_3, u_4,$	$u_5, u_6, u_7, u_8, u_9,$	u_{10}, u_{11}
`DFS`	$R, u_3, u_5, u_7, u_9,$	$u_{11}, u_8, u_2, u_4, u_6,$	u_{10}, u_1
`BFS`	$R, u_3, u_2, u_4, u_1,$	$u_5, u_9, u_8, u_6, u_7,$	u_{11}, u_{10}
`Lyr`	$R, u_1, u_3, u_2, u_4,$	$u_5, u_6, u_9, u_8, u_7,$	u_{11}, u_{10}
`Lev`	$R, u_3, u_2, u_1, u_5,$	$u_4, u_8, u_7, u_6, u_{10},$	u_{11}, u_9
`Ly2`	$R, u_2, u_3, u_1, u_4,$	$u_5, u_6, u_7, u_9, u_8,$	u_{10}, u_{11}
`GrB`	$R, u_3, u_2, u_1, u_4,$	$u_5, u_7, u_9, u_8, u_6,$	u_{11}, u_{10}
`GrD`	$R, u_3, u_2, u_5, u_7,$	$u_9, u_1, u_4, u_8, u_{11},$	u_6, u_{10}

Figure 6.5 shows an example of clustering of modifications according the different techniques. In this case, each disk page can contain at most five modifications. In this specific configuration most sorting techniques (`Err`, `BFS`, `Lyr`, `Ly2` and `GrB`) give the same clusters as results.

Note that a clustering technique which simply sorts modifications and fulfills each disk block (or stripe) with a contiguous set of sorted modifications does not introduce any overhead in storage space. As mentioned in Section 6.1.2, a disk block is usually at least 4 KBytes. There is no upper bound in block size, but a block that spans on a whole disk track bests amortize seek times and latency, and, thus, there is no need to create larger blocks. This results in an upper bound equal to 50 to 200 KBytes, according to disk radius and density. In a RAID subsystem with up to 5 disk configured for disk striping, it can result in up to 200 to 1000 KBytes.

If we consider an explicit 3D MT, each modification requires on average 325 Bytes. This results in a block transfer size B in a range between 12 and 150-600 modification, if we consider a single disk architecture and between 60 to 600-3000 with a large RAID subsystem.

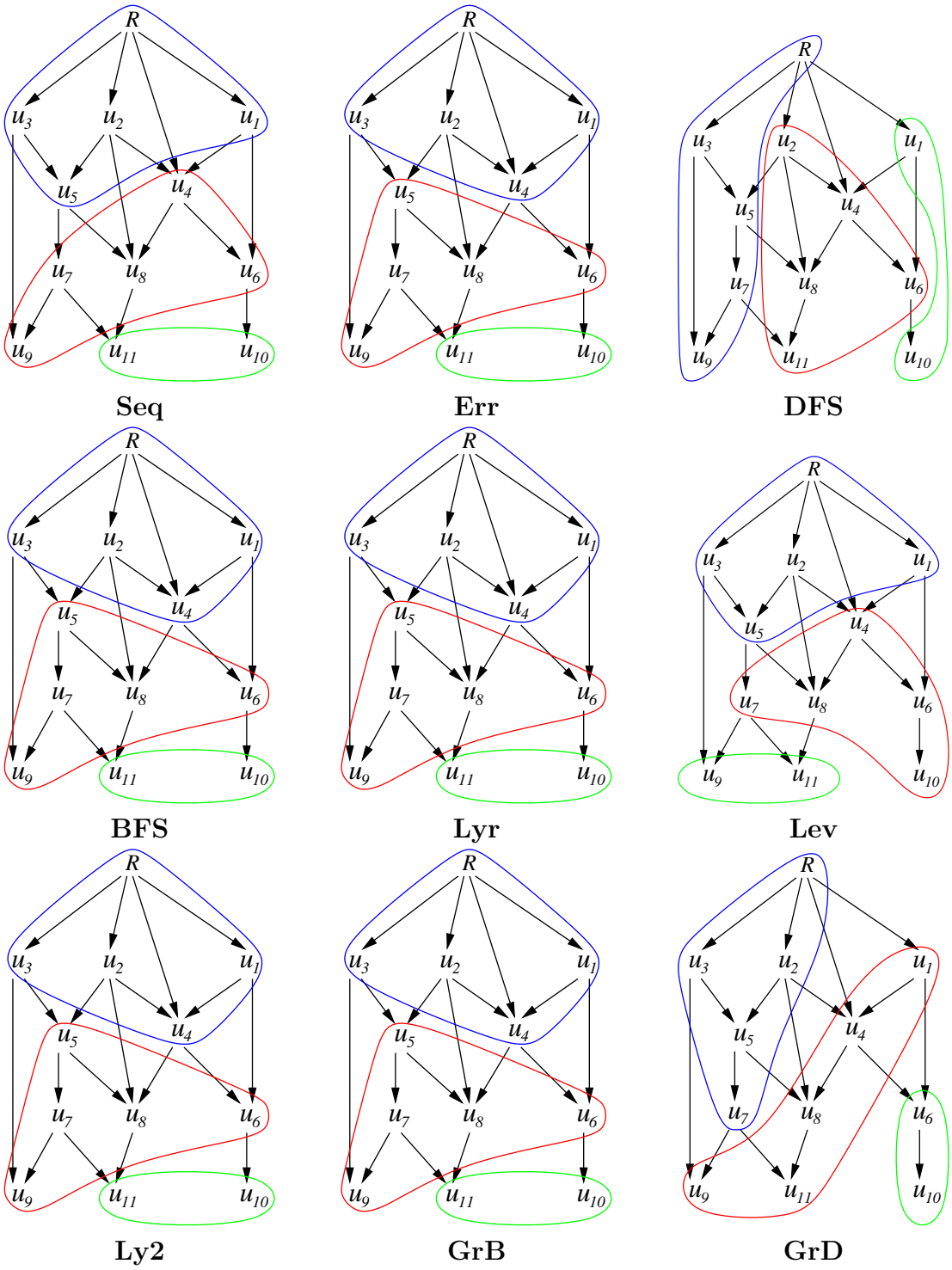


Figure 6.5: Cluster of modifications according to sorting criteria. Each cluster can contain at most five modifications

The number of modifications per block can dramatically increase by a compact encoding for the modifications, but also decreases if we decide to store additional information, such as bounding box, approximation error, in order to speed up the execution of primitive IS_ACCEPTABLE.

6.3.3 Space partitioning of an MT

Queries that select a certain region of interest suggested to subdivide the multi-resolution model according to a space partitioning technique. Spatial coherent regions seem to better capture the locality of queries involving a region of interest, as, for instance, in view-dependent rendering.

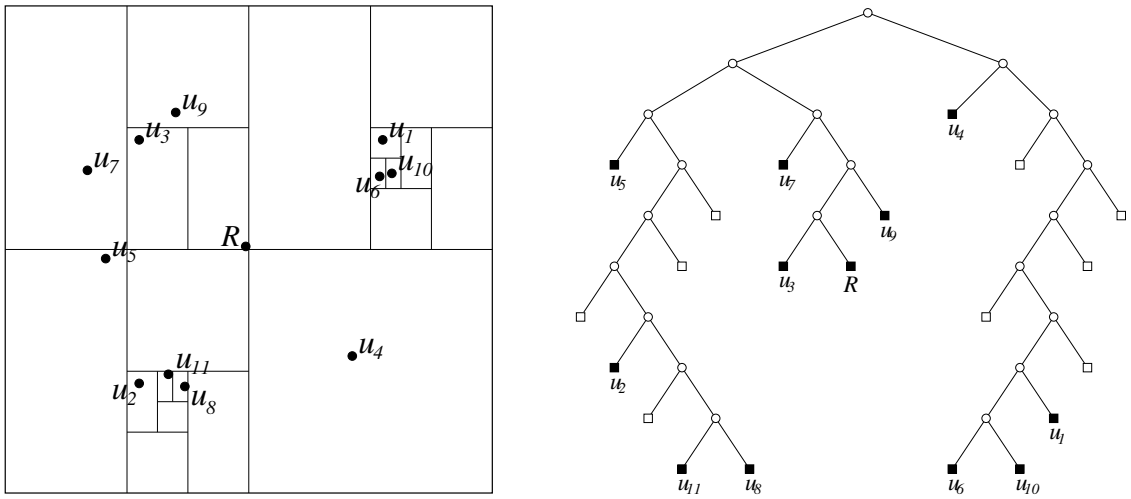
A geometric dataset usually is characterized by a variable resolution, since some portion of the domain are more finely tessellated. This imposes the use of an adaptive partitioning scheme. To apply a space partitioning technique, we associate a single point in the Euclidean space with each modification in the MT (for instance, the centroid of the modification). Thus, we have considered spatial indexes for points in the Euclidean space. A complete analysis and survey of such techniques is presented in [Sam05].

We consider at first a common spatial index, that is the *Point Region quadtree* (PR quadtree) [Ore82, Sam90]. It is a spatial index for points in the two-dimensional Euclidean space, based on the recursive decomposition of a square domain in $\mathbb{E}^2$, which contains the data points, into four quadrant obtained by splitting the square block through the midpoint of the square. The subdivision is performed recursively until a full block contains just one point. Thus, the data points are in the leaves of a PR-quadtree and the internal nodes are just discriminators. A PR-quadtree can be extended to d-dimensional space. In this case, each recursive decomposition splits a block into 2^d blocks. But a PR-quadtree suffers the curse of dimensionality, since it performs well in low dimensional space, very large and a high percentage of the leaves is empty. In $\mathbb{E}^2$, at least 25% of the leaves are full, while in $\mathbb{E}^4$ it decrease to 6.25%.

Since the Multi-Tessellation is a data structure that can handle multiresolution model in arbitrary dimensions, we have been considering a space partitioning technique that does not depend on space dimensionality. Thus, we have selected *Point Region k-d trees* (PR k-d trees) [Ben75] and *R* trees* [BKSS90].

Subsection 6.3.4 describes the PR k-d tree and our application to the Multi-Tessellation. After the space partitioning step, in order to store modifications on disk block, we need to group them. In Subsection 6.3.5 we describe our grouping technique which is based on a *PK tree* [WYM98].

The previous techniques are designed for space partitioning of point data sets. Since



(a) (b)

Figure 6.6: A PR k-d tree with the resulting partition of space (a), and the tree representation (b)

the MT is composed of modifications, we also experimented an object-based partitioning technique, an R*-tree policy. This is described in Subsection 6.3.6.

6.3.4 Partitioning an MT through the PR k-d tree

A PR k-d tree is a spatial index based on the recursive subdivision of the domain containing the data points. At each step the domain is halved. The halving process cycles through different dimensions in a predefined and constant order. At each step a block is subdivided in its mid-point. The PR k-d tree is well suited for the MT since it scales well to higher dimensional spaces.

In the construction of the PR k-d tree we have considered the dimensions of the Euclidean space in which the MT is embedded (2D or 3D in most cases). For a three-dimensional space we considered the standard order x, y, z. We split at first level along x, at the second level along y, at the third level along z, and then the process restarts from x.

We associate with each modification its centroid, and we construct the PR k-d tree according to the coordinates of centroids. We have performed several tests in higher dimensions by adding the approximation error $e(u)$ or the layer $l(u)$ or the distance from the root $d(u)$ as an additional dimension. If we consider a three dimensional model and an additional parameter, we construct a four dimensional PR k-d tree with the additional parameter as first halving criterion and then all the coordinates in the standard order.

Figure 6.6 (a) shows the space partition generated by a PR k-d tree in dimension 2. The corresponding tree is shown in Figure 6.6 (b). White squares represent empty leaves. It is easy to notice that a drawback of the PR k-d tree is that it can result in an unbalanced tree if data are not uniformly distributed: this fact is well known for all point-region trees (see [Sam05]). The problem can be partially overcome by the grouping step performed by the PK tree described in Section 6.3.5.

6.3.5 Grouping a PR k-d tree through a PK tree

To store a PR k-d tree on disk, we could have used a technique commonly used for quadtrees and octrees, which consist of computing the location codes of the full leaves in the PR k-d tree and storing the location codes in a B-tree [Sam05]. Such technique has the disadvantage that the boundary of the domain covered by leaves stored in a node of the B-tree can be arbitrary complex. In general, the domain might be not connected and, thus, we could loose spatial coherence.

An interesting alternative consists of using a PK-tree as a grouping mechanism. Originally proposed in [WYM98], a PK-tree is clustered by a parameter k, called the *instantiation value*, which is the minimum number of nodes in the tree grouped in a cluster. A PK tree is constructed by applying a bottom-up grouping process to the nodes of a tree T. Nodes belonging to T are grouped into clusters until the minimum occupancy k has been reached. During the grouping process empty leaves of the tree T are removed. If the tree T is an n-ary tree, each node, except the root, has a minimum of k children or objects and a maximum of $n \cdot (k-1)$.

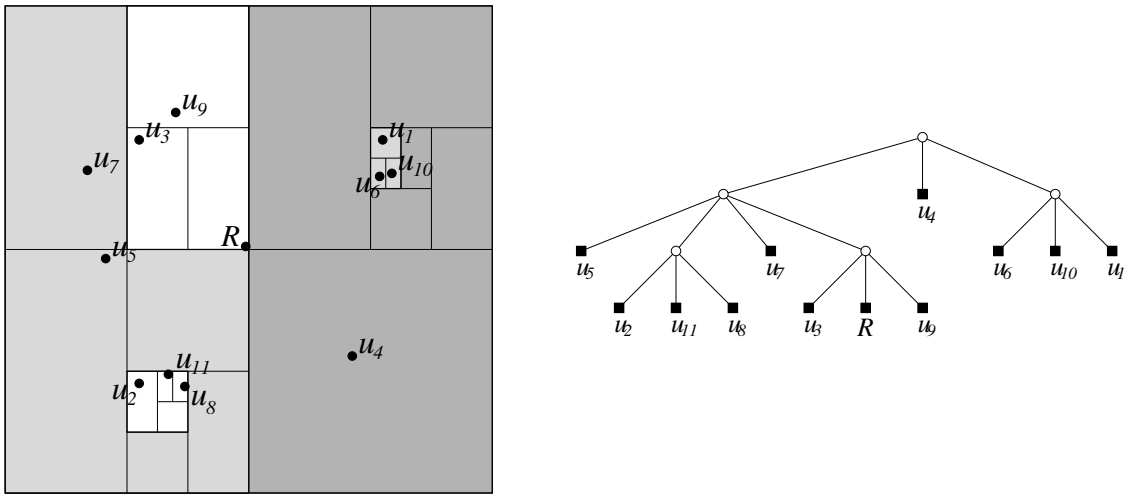
Since the PR k-d tree is a binary tree, an interesting property of the PR k-d tree grouped according to the PK-tree, that we called a *PK PR k-d tree*, is that each node has a minimum of k children and a maximum of $2 \cdot (k-1)$, regardless of the dimensionality of the space [WYM98]. This guarantees that disk blocks are at least half-full.

Figure 6.7 shows the PK PR k-d tree corresponding to the PR k-d tree presented in Figure 6.6.

6.3.6 Grouping modifications based on an R*-tree

R-trees are an object based-technique that performs object aggregation [BKSS90]. In an R-tree, each object is defined by its bounding box. The first step aggregates up to k bounding boxes into a box of minimum size that contains them. This process is repeated recursively until there is just one block left. Each box is mapped to a disk block.

One of the intrinsic problems with such techniques is that boxes can overlap, and the com-



(a) (b)

Figure 6.7: A PK PR k-d tree with $k = 3$ corresponding to the PR k-d tree shown in Figure 6.6: the resulting partition of space (a), and the tree representation (b). Bold lines indicate the boundaries of blocks corresponding to internal nodes. Gray levels represent the layer in the tree.

putation of minimum size boxes (which minimize domain coverage) is not trivial. Several extent-based aggregation techniques have been developed. The main differences among such techniques are related to coverage and to minimization of overlapping areas. R*-trees [BKSS90] have been shown to be the best extent-based aggregation technique. They can handle d-dimensional objects, and minimize domain coverage and rectangle overlap.

In our tests we used have the R*-tree library developed by Bernhard Seeger, combined with our prototype. We associate a minimum axes-aligned bounding box with each modification. If we consider another parameter, such as approximation error, we consider an higher dimensional bounding box taking into account the additional dimension.

6.3.7 Clustering techniques

Our prototype has been designed to compare the efficiency of different clustering techniques, in order to be able to implement the most efficient out-of-core multiresolution data structure. Since geometric computation are not affected by the out-of-core design, we focus our investigations on the I/O operations performed by the selective refinement algorithm. In order to achieve this result, we have implemented clustering techniques based on total sorting of modification and/or on space partitioning, as described in Sections 6.3.2 and 6.3.3.

The simplest clustering techniques are based on a total sorting of modifications. Computation of clusters based on a sorting of modifications requires two steps:

- modifications are sorted according to one of the criteria described in Subsection 6.3.2;
- starting from the beginning of the ordered set, we store modifications into disk pages until each disk page is completely full.

Space consumption of these techniques is optimal. If our multiresolution model consists of N items (i.e. modifications), we use $\lceil \frac{N}{B} \rceil$ disk blocks if $D = 1$, or $\lceil \frac{N}{B \cdot D} \rceil$ stripes in a RAID system.

We have also applied a space partitioning and grouping technique, and store in each disk block a cluster of modifications. Our prototype can compute both a PK PR k-d tree, and an R*-tree.

According to Subsection 6.3.4, a PK PR k-d tree can be built adopting the space coordinates of the centroids and, optionally, other criteria like the approximation error, or the layer. This gave us the possibility to test several clustering policies based on PK PR k-d tree. In particular we have implemented techniques based on:

- Cartesian coordinates of the centroid (it will be labeled `PKTree`);
- coordinates plus approximation error $e(u)$: `PKTreeErr`;
- coordinates plus layer $l(u)$: `PKTreeLyr`;
- coordinates plus distance $d(u)$: `PKTreeLy2`;
- coordinates plus level $L(u)$: `PKTreeLev`.

The same flexibility is offered by an R*-tree. In our prototype we have implemented three and four dimensional R*-trees, based on Cartesian coordinates and, optionally, another criterion, like the approximation error or the layer. We have implemented the following techniques, based on R*-tree:

- Cartesian coordinates (it will be labeled `RTree`);
- coordinates plus approximation error $e(u)$: `RTreeErr`;
- coordinates plus layer $l(u)$: `RTreeLyr`;
- coordinates plus distance $d(u)$: `RTreeLy2`;

- coordinates plus level $L(u)$: `RTreeLev`.

We have combined a sorting criterion on with a space partitioning technique. The idea is to combine a technique that subdivides the DAG in subsets of modification according to their depth, with a technique that subdivides the space. At a lower resolution, we are interested to have clusters that span the whole domain, while, as the resolution increases, we are looking for clusters associated with finer space subdivisions.

In order to achieve this goal, we have adopted a technique that interleaves the effect of a sorting rule and the effect of a space partitioning rule similar to a PR k-d tree.

We consider the sequence of modifications in an MT generated by any of the sorting rules, described in Subsection 6.3.2. We store in a disk page the first k modifications in the sequence, then partition the remaining modifications according to the space partitioning criterion. The process is repeated recursively on each subset of modifications, and generates a tree structure.

In this data structure we perform a recursive subdivision of the domain containing the centroids or the domain defined by the centroids plus an additional parameter, as in the PR k-d tree, since we subdivide the domain in half. The halving process cycles through different dimensions in a predefined and constant order. The difference, compared to a PR k-d tree, is that each node of the tree can store up to k modifications. The resulting data structure is a binary tree in which each node corresponds to a disk block.

The algorithm that we use can be summarized as follows:

1. compute a sorted sequence of modifications. Let L_S be such a sequence;
2. let B be the first block;
3. remove first k modifications from L_S and store them with the current disk block B;
4. if L_S is not empty, L_S is split into two sublist L'_S and L''_S according to the halving criterion adopted by the PR k-d tree;
5. apply 3 recursively on L'_S and L''_S.

Figure 6.8 shows an example of this technique when $k = 5$. We have sorted the modifications according to approximation error order, and we get $L_S = [R, u_1, u_2, u_3, u_4, u_5, u_6, u_7, u_8, u_9, u_{10}, u_{11}]$. The first disk page contains the root R plus modification u_1, u_2, u_3, u_4. L_S becomes $[u_5, u_6, u_7, u_8, u_9, u_{10}, u_{11}]$ from which we extract $L'_S = [u_6, u_{10}]$ and $L''_S = [u_5, u_7, u_8, u_9, u_{11}]$. List L'_S can fit in one disk page. Thus, the process restarts with L''_S. Also L''_S can fit in a page. In this case, the partitioning technique is optimal in the number of disk pages.

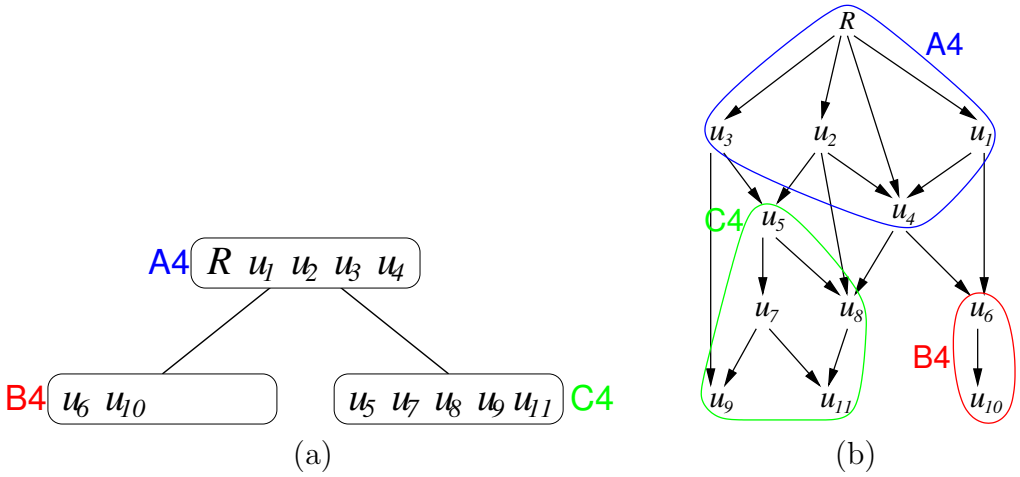


Figure 6.8: Clustering obtained mixing sequential sorting plus a space partitioning as the one adapted PR k-d tree. Each disk page can fit at most five modifications.

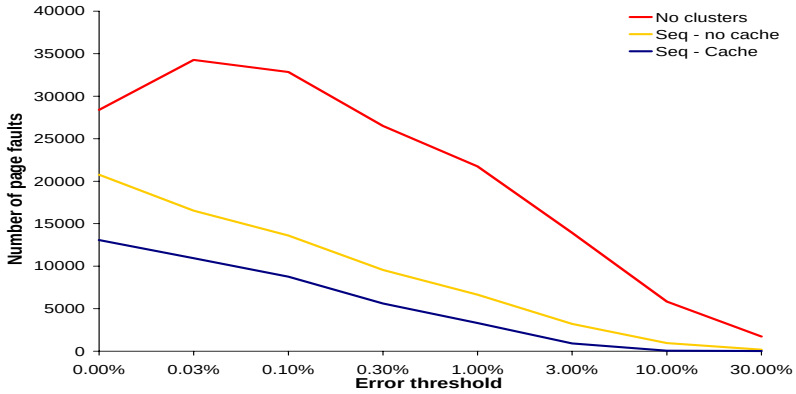
This technique, that combines sorting with space partitioning, has been applied by using all sorting criteria we developed, that are described in Section 6.3.2, and the five halving rules adopted for the PR k-d tree, described in Subsection 6.3.4.

6.4 Caching policies

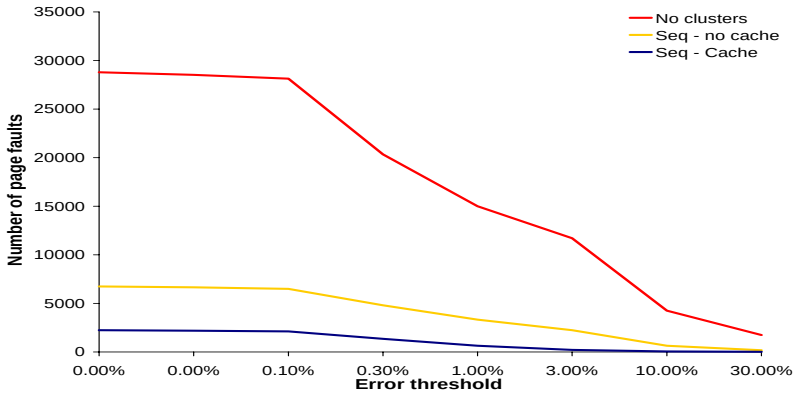
During selective refinement, we need to load clusters of modifications in main memory. Sometimes a cluster can be referred more than once, especially if we are building large clusters. This suggests to develop a buffer for clusters just loaded from disk. A common graphics workstation has between 1 and 4 GBytes of main memory (this means it can hold between 3 and 12 million of modifications). Even if a huge multiresolution model can be several times larger, it is useful to use part of the main memory as a cache for clusters of modifications. In our prototype it is possible to have a cluster cache, whose size can be defined by a user, and can be dynamically redefined.

If we are loading from disk a new modification cluster, and the cache is full, we should remove one of the clusters already cached. We have implemented two different cache replacement policies: oldest and less-recently-used. According to our tests the less-recently-used policy seems to offer better results. Every test reported in Section 6.5 is based on this latter cache replacement policy.

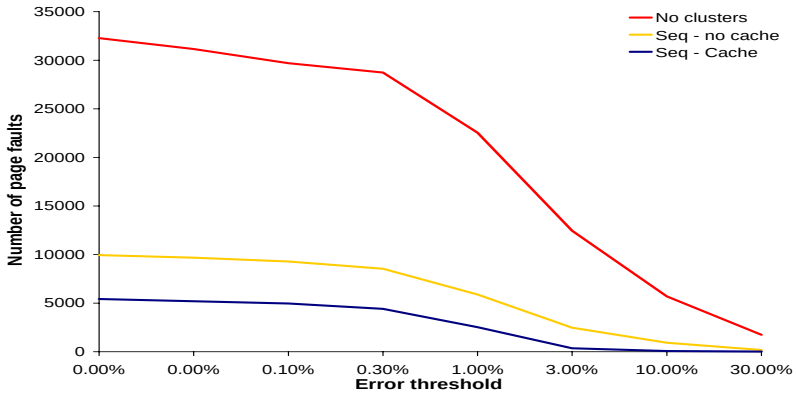
Figure 6.9 shows the effect of clustering based on sequential sorting, and of a small cache that can fit up to 1% of the modification of a multiresolution tetrahedral model. Tests have been performed on the San Fernando dataset. Every graph reports the number of



(a)



(b)



(c)

Figure 6.9: Number of I/O operations for a 3D MT stored on external memory. We used the explicit MT data structure. Disk accesses without clustering modifications (red), with sequential clustering (yellow), with clustering and caching (blue). (a) Query at uniform LOD (b) Query at variable LOD, based on a region of interest (c) Query at variable LOD, based on iso-contours

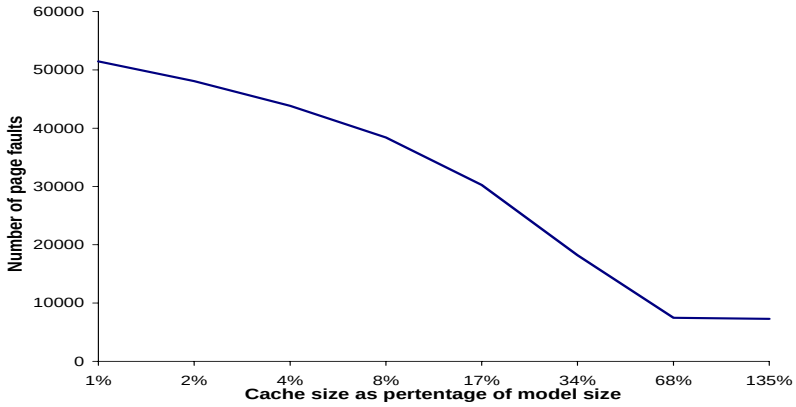


Figure 6.10: Number of page faults when performing a query at uniform resolution with different cache size.

I/O operations for an MT stored on external memory as a function of the error threshold. The red graph represents disk accesses without clustering modifications, the yellow one represents disk accesses with sequential clustering and no caching, the blue graph is based on the same clustering technique and an additional cache that can fit up to 11% of the whole model. The graph in Figure 6.9 (a) is obtained by performing selective refinements at uniform LOD, the graph in Figure 6.9 (b) presents results with variable LOD (query based on a region of interest), while the in Figure 6.9 (c) presents results with LOD queries based on iso-contours.

Figure 6.10 shows the relation between cache size and number of page faults in a multiresolution model partitioned according to the `GrD` clustering rule. Recall that `GrD` clustering simulates the selective refinement at uniform LOD by using a DAG traversal strategy based on depth-first search.

6.5 Analysis and comparisons

In this Section, we present an exhaustive comparison among the different clustering techniques. Our tests focused on MT for volume data sets. Unfortunately, up to now, no out-of-core simplification tools is available for tetrahedral datasets. For this reason, we have used the largest model that can be handled by an in-core, high-quality, simplification tool [CCM+00]. In order to get reasonable and meaningful results, we have artificially reduced the amount of available core memory, in order to force the out-of-core prototype to load only a subset of disk blocks.

In our prototype we have implemented more than sixty clustering techniques. For this reason, we have organized the results in several graphs. The best techniques in each subset, corresponding to a graph, are further compared. Each graph represents a ratio between the number of disk accesses required to perform a set of selective refinement queries and the number $n = N/B$ of disk blocks required to store the model. Recall that N is the size of the data, i.e. the number of modifications associated with an MT, and B is the size of a disk page, i.e. the number of modifications that can fit in a disk page. This allow us to compare results, independently on model size.

We show results mainly on San Fernando data set. Figure 6.11 shows the dataset at full resolution. The dataset consists of 378,747 points and the mesh at full resolution contains 2,067,739 tetrahedra. Figure 6.11 compares the different clustering techniques by using extractions at uniform resolution.

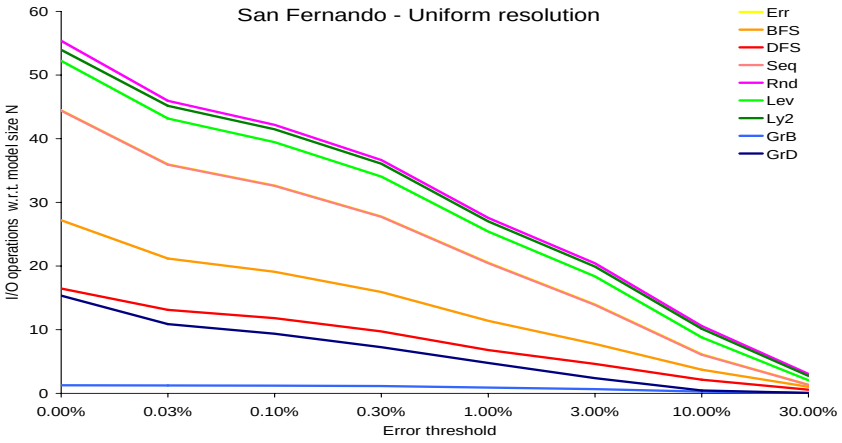
Figure 6.11 (a) compares results obtained with nine sorting rules defined in Subsection 6.3.2. Cache size is set to 1% of the size of the model, in order to enhance differences among clustering techniques. Random clustering `Rnd` offers the worst performances. A query at uniform resolution with error threshold set to zero requires a number of disk access equal to 56 times loading of the whole multiresolution model. Sorting based on the error threshold (`Err`) and the order induced by simplification (`Seq`) give the same results. `DFS` and `BFS` perform quite well. The best results are obtained by breadth-first visit with inclusion of ancestors, according to the order induced by the procedure FORCE_REFINE (`GrB`).

The graph in Figure 6.11 (b) compares results obtained with clustering techniques that combine sorting and space partitioning. Space partitioning is based on the coordinates of the centroid associated with each modification. The space partitioning improves worst sorting techniques but it almost does not affect the ratio among different sorting techniques. Sequential sorting coupled with space partitioning performs best, and the same happens to distance sorting `Ly2`.

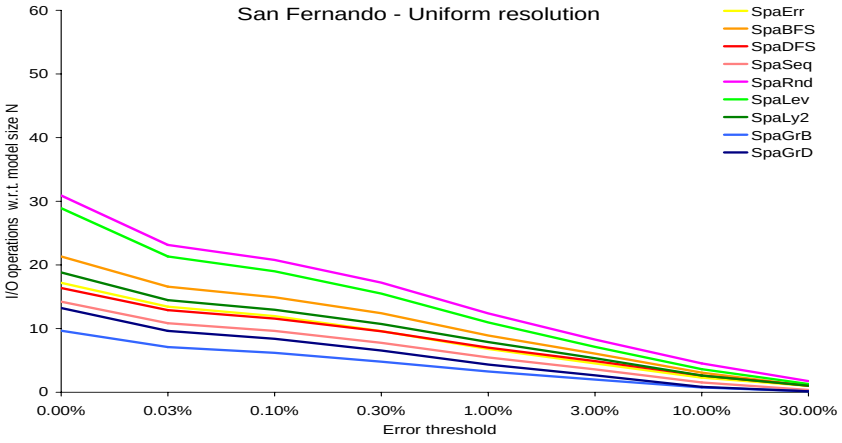
The graph in Figure 6.11 (c) compares results obtained with clustering techniques that combine the sorting rules with space partitioning according to the approximation error and the three coordinates. The subdivision rule in the PR k-d tree subdivides first according to the approximation error and then according to the x, y, z coordinates of the centroid. Results are similar as the ones shown in Figure 6.11 (b).

The graphs in Figure 6.12 (a), (b) and (c) compare results obtained with clustering techniques that combine the sorting rules with space partitioning according to the level (a), distance (b) or layer (c), respectively, and the three coordinates. In such cases the influence of sorting rules is really low.

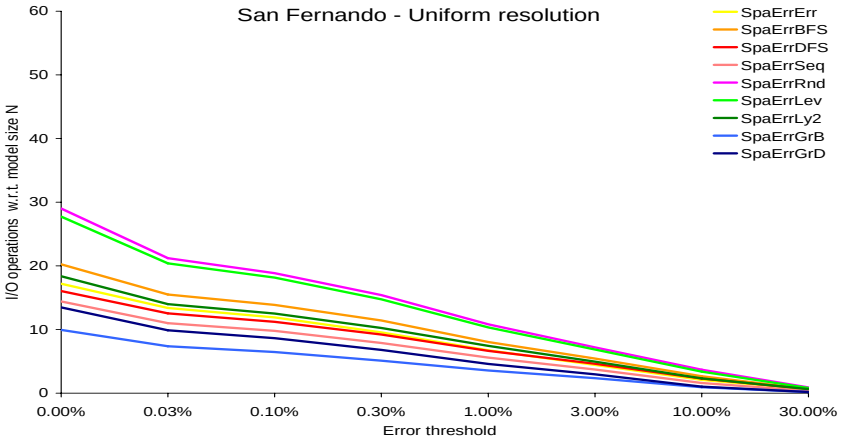
The graph in Figure 6.13 (a) compares the results obtained by using the PK PR k-d tree built with different space partitioning rules, namely:



(a)

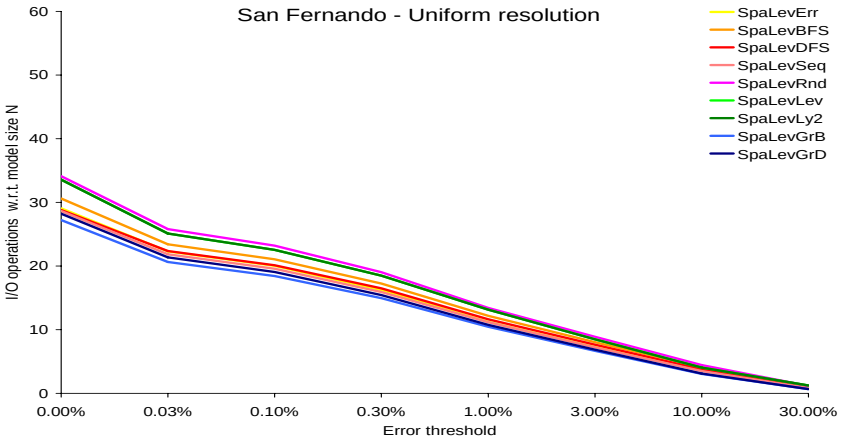


(b)

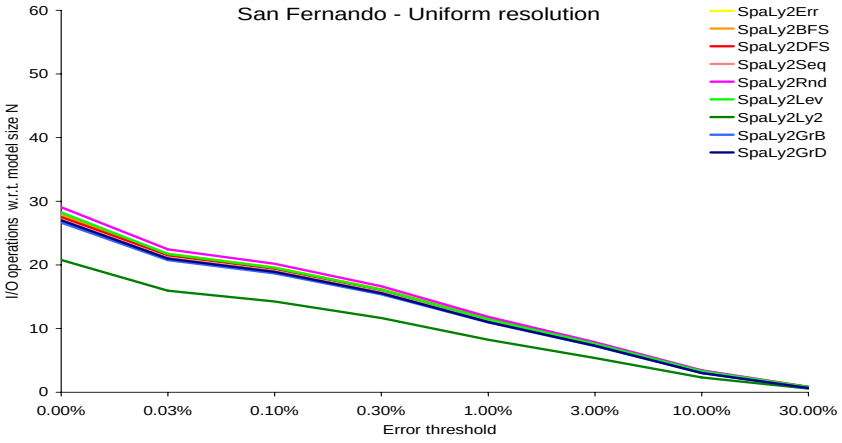


(c)

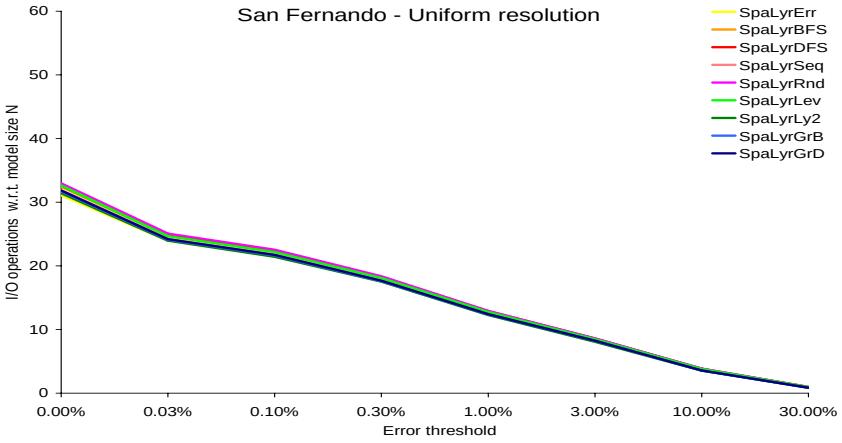
Figure 6.11: Number of I/O operations performing queries at uniform resolution on San Fernando dataset - part 1/3.



(a)

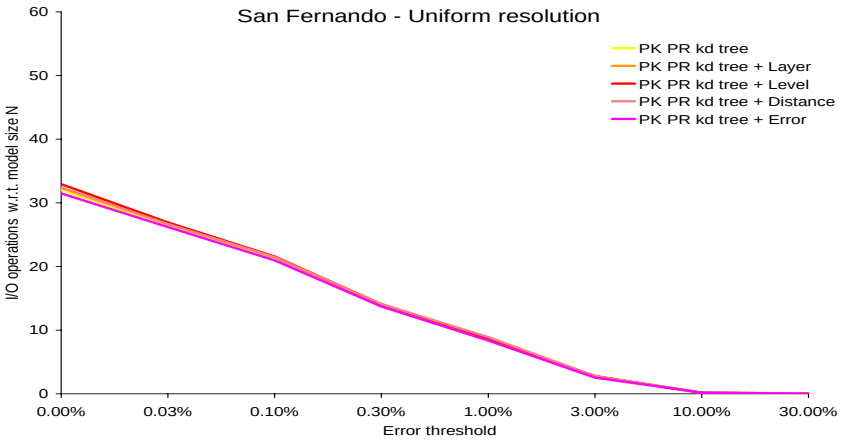


(b)

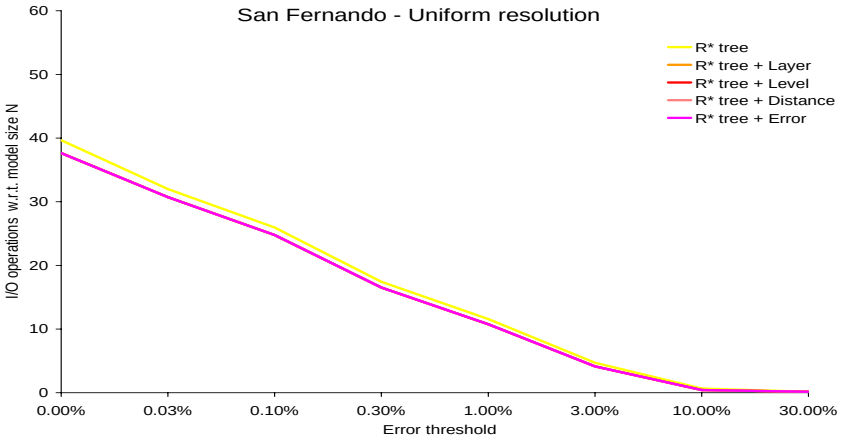


(c)

Figure 6.12: Number of I/O operations performing queries at uniform resolution on San Fernando dataset - part 2/3.



(a)



(b)

Figure 6.13: Number of I/O operations performing queries at uniform resolution on San Fernando dataset - part 3/3.

- coordinates of the centroid (`PK PR kd tree`);
- coordinates plus the layer (`PK PR kd tree + Layer`);
- coordinates plus the level (`PK PR kd tree + Level`);
- coordinates plus the distance (`PK PR kd tree + Distance`);
- coordinates plus the error (`PK PR kd tree + Error`).

Results are very similar, but partitioning with error plus coordinates performs slightly better.

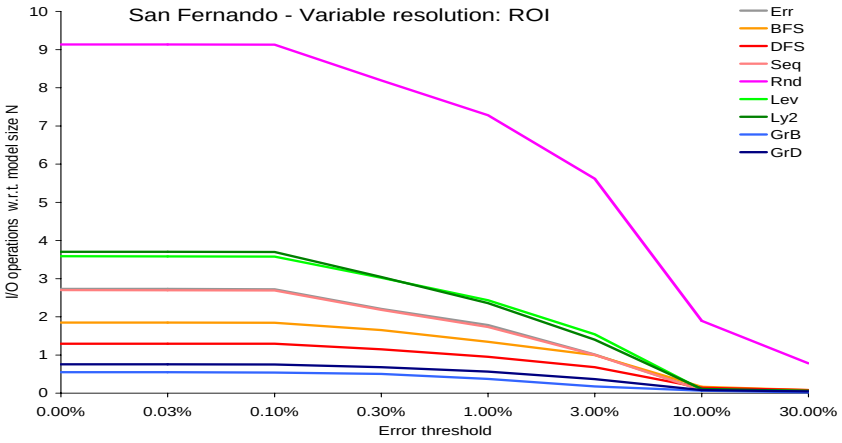
The graph in Figure 6.13 (a) compares results obtained by PK PR k-d tree built by partitioning space according the three coordinates and adding the level, distance, layer or the approximation error. Results are very similar, but partitioning with error plus coordinates performs slightly better.

The graph in Figure 6.13 (b) shows the same comparisons as in Figure 6.13 (a), replacing the PK PR k-d tree with an R* tree. In this case, a bounding box is used instead of the centroid. In this case as well, results are very similar, but partitioning with error plus coordinates performs slightly better.

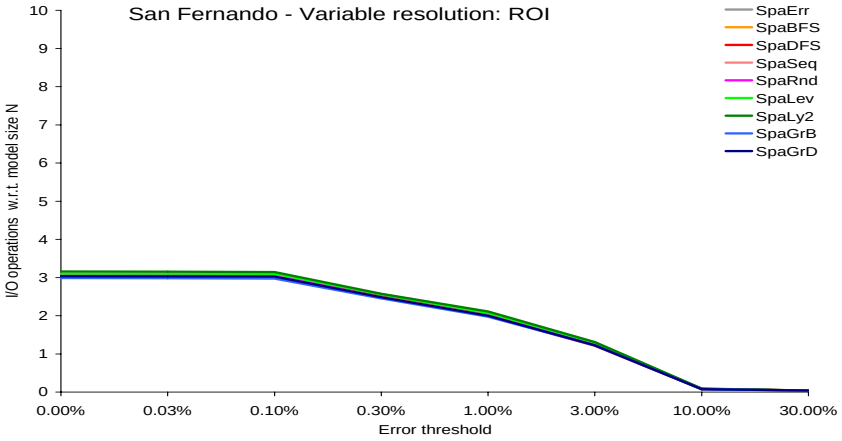
Graphs in figures 6.14, 6.15 and 6.16 describe the behavior of clustering techniques with a query at a variable resolution with a box as a focus set. We fix the error threshold inside the box, while it is arbitrary large outside the box.

Figure 6.17 (a) compares the best clustering techniques presented in Figure 6.11, 6.12 and 6.13. We consider the following techniques:

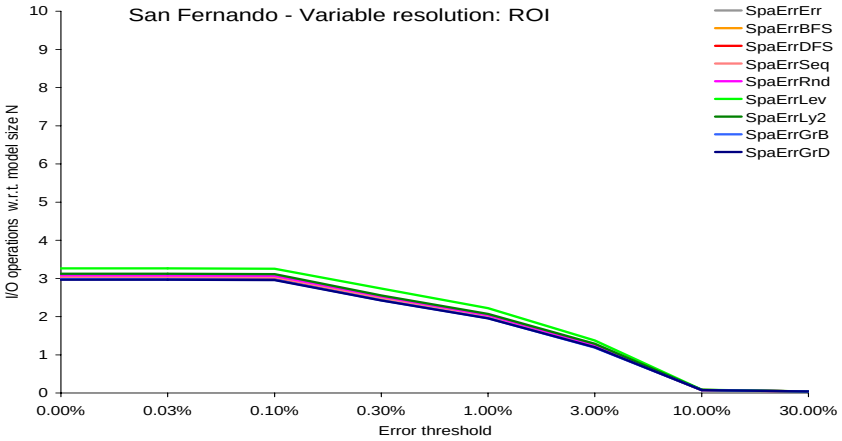
- sequential sorting (`Seq`);
- sorting according to approximation error (`Err`);
- sorting according to depth-first traversal of the DAG (`DFS`);
- sorting according to breadth-first traversal of the DAG and insertion of ancestors (`GrB`);
- sorting according to depth-first traversal of the DAG and insertion of ancestors (`GrD`);
- space partitioning on three coordinates, and `GrB` sorting (`SpaGrB`);
- space partitioning on approximation error and three coordinates, and `GrB` sorting (`SpaErrGrB`);



(a)

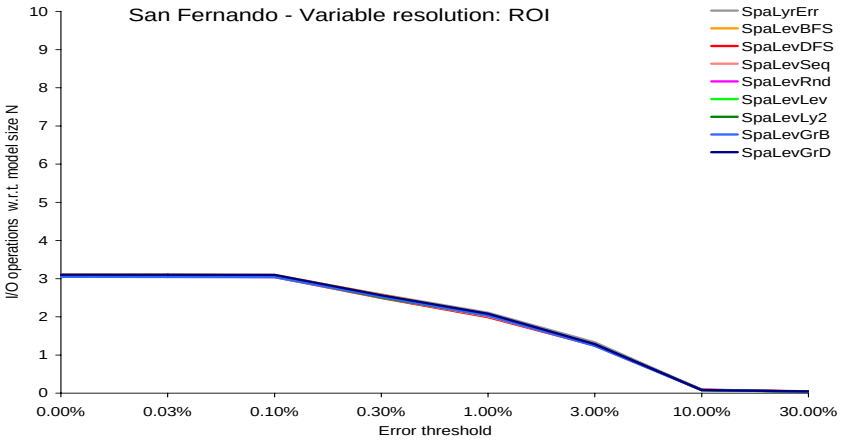


(b)

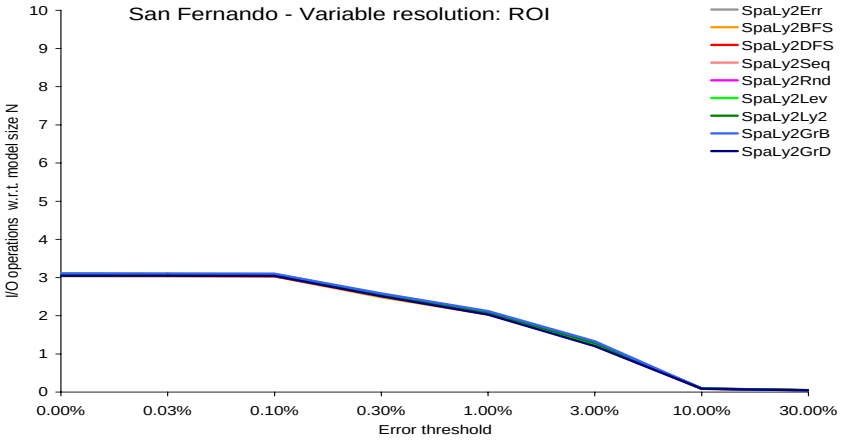


(c)

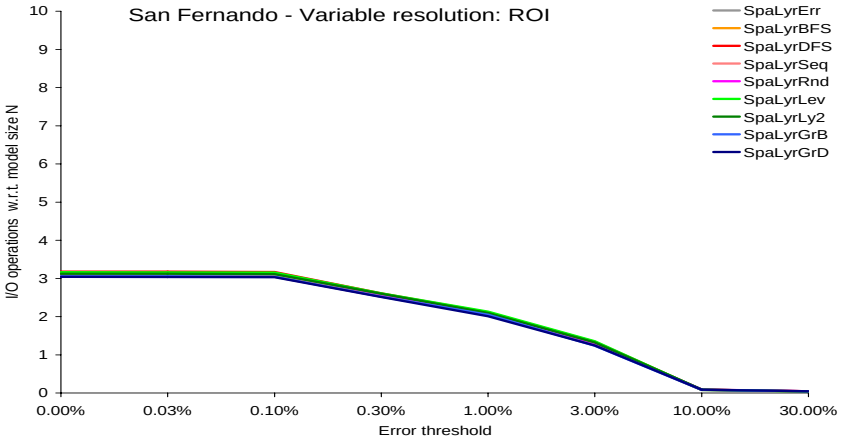
Figure 6.14: Number of I/O operations performing queries at uniform resolution on San Fernando dataset - part 1/3.



(a)

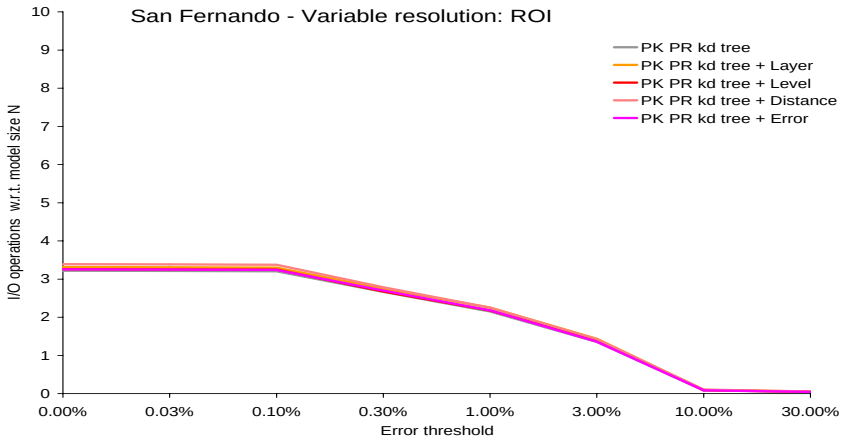


(b)

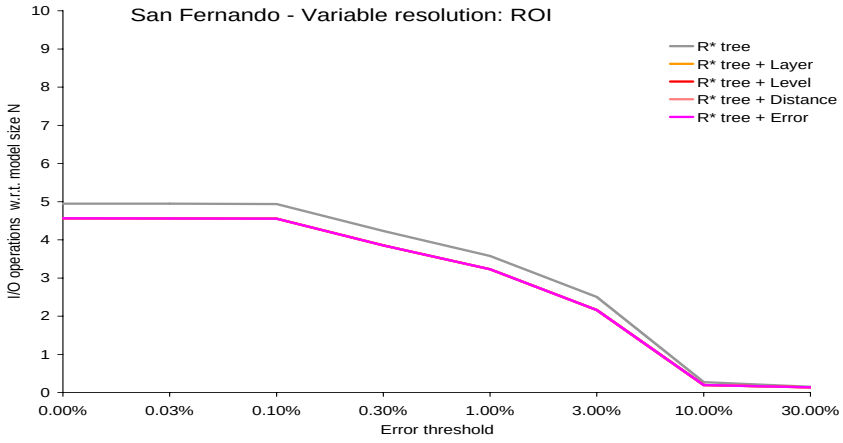


(c)

Figure 6.15: Number of I/O operations performing queries at uniform resolution on San Fernando dataset - part 2/3.

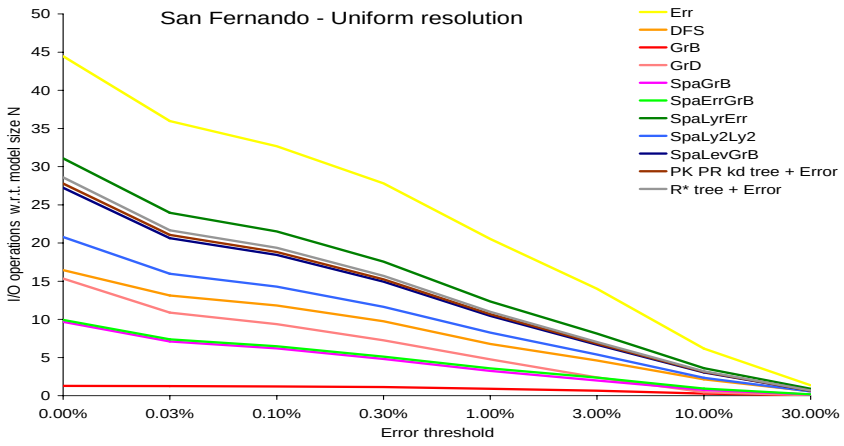


(a)

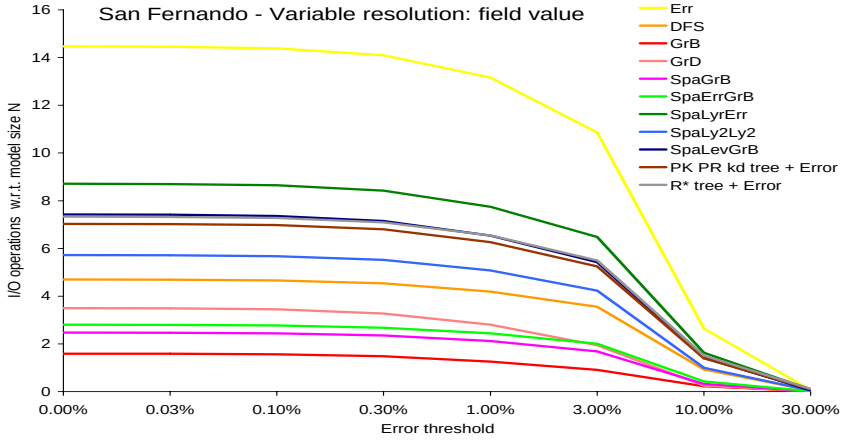


(b)

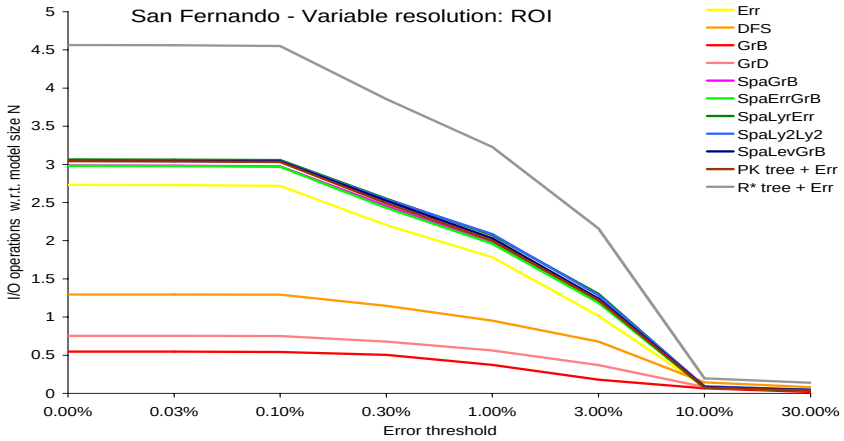
Figure 6.16: Number of I/O operations performing queries at uniform resolution on San Fernando dataset - part 3/3.



(a)



(b)



(c)

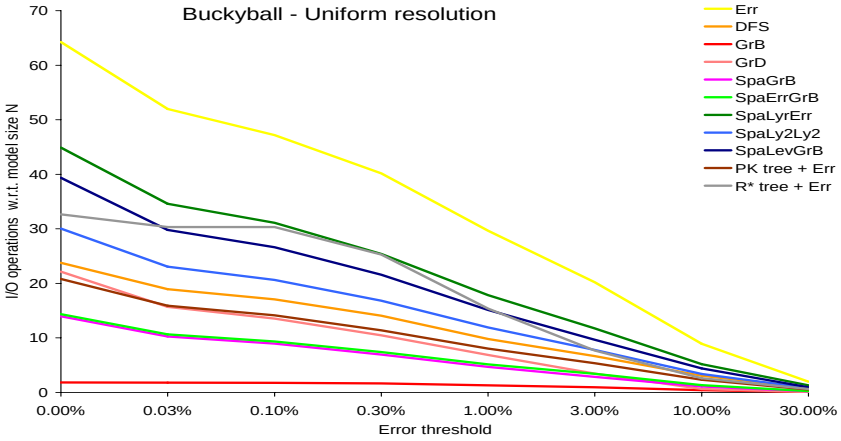
Figure 6.17: Number of I/O operations performing queries at uniform resolution (a), variable resolution with a field value as focus set (b) or variable resolution with a box as focus set (c), on San Fernando dataset.

- space partitioning on layer and three coordinates, and `Err` sorting (`SpaLyrErr`);
- space partitioning on distance and three coordinates, and `Ly2` sorting (`SpaLy2Ly2`);
- space partitioning on level and three coordinates, and `GrB` sorting (`SpaLevGrB`);
- space partitioning on approximation error and three coordinates, and grouping with PK tree (`PK PR kd tree + Error`);
- space partitioning and grouping with an R*-tree built according with approximation error and three coordinates (`R* tree + Error`).

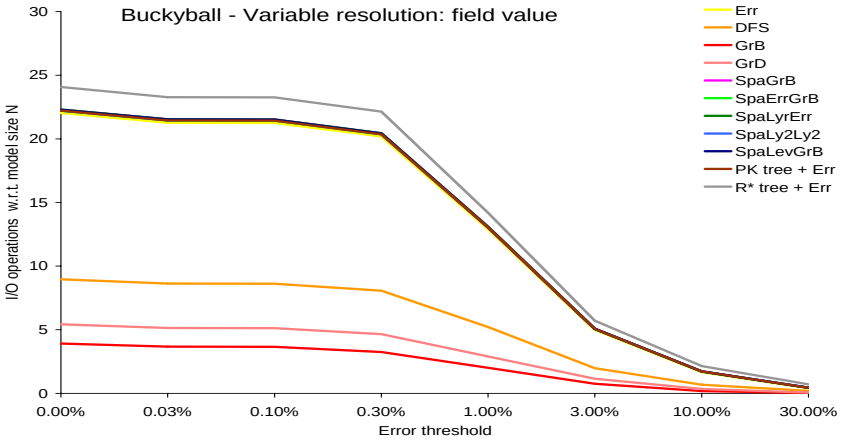
It is easy to notice that `GrB` outperforms other clustering techniques. With selective refinement queries at uniform resolution this result is quite obvious, since it cluster modifications in the same order of a selective refinement traversal at uniform resolution. It is much more interesting to note that `GrB` sorting, even with queries at variable resolution, offers excellent results. Results on selective refinement at variable resolution with a field value as focus set are shown in Figure 6.17 (b), while results on selective refinement at variable resolution with a box as focus set are shown in Figure 6.17 (c). It is also interesting to note that, even with a really small cache, that is about 1% of the size of the whole model, a clustering technique based on `GrB` exhibits really small overhead, compared to loading of the whole model.

We performed the complete set of test also with the Buckyball dataset. In this case, every space partitioning or space partitioning coupled with sorting rules behave nearly in the same way. The difference among the different techniques is less than 5%. For this reason, we have shown in our graph only one of them. This behavior is probably related to the regular structure of Buckyball dataset. Figure 6.18 shows the results of clustering techniques based on sorting. A space partitioning based on PR k-d tree coupled with `Err` sorting and R* tree partitioning based on a four-dimensional bounding box with approximation error and vertex coordinates.

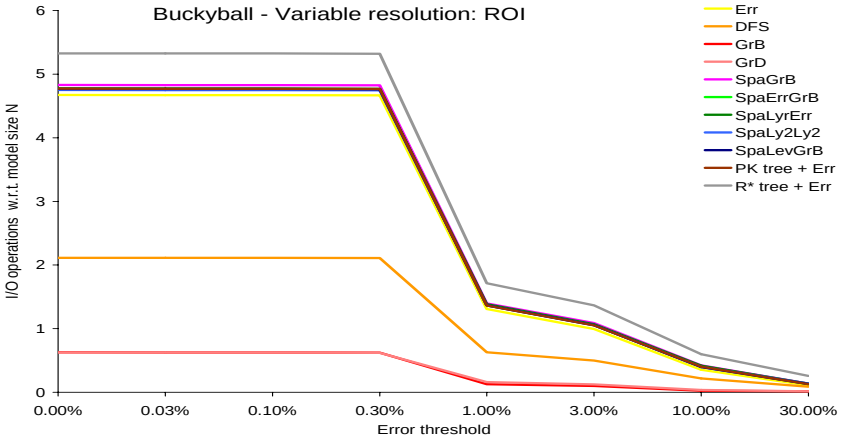
Figure 6.18 (a) shows the behavior of clustering techniques with queries at a uniform resolution. Results are coherent with the behavior of clustering techniques in the San Fernando dataset. `GrB` is still the best approach. Figure 6.18 (b) shows the behavior of clustering techniques with queries at variable resolution with a field value as focus set. Results are coherent to the behavior of clustering technique in the San Fernando dataset. Figure 6.18 (c) shows the behavior of clustering techniques with queries at variable resolution by using a region of interest with high resolution inside a box, arbitrary large elsewhere. The only difference we can observe comparing with the previous results, `GrB` and `GrD` performs nearly in the same way. It is also interesting to note that with this kind of queries, even with an error threshold set to zero, the number of disk reads with the out-of-core structure is lower than the number of disk reads required to load the multiresolution model in an in-core architecture.



(a)



(b)



(c)

Figure 6.18: Number of I/O operations performing queries at uniform resolution (a) or variable resolution with a field value as focus set (b) or variable resolution in a ROI (c) on Buckyball dataset.

`GrB` is clearly the best technique implemented in our prototype. Its overhead is small or null, and it scales well based on the cache size. The only drawback is that it requires a traversal of the dependency relation that mimics the selective refinement algorithm. This visit has to be performed out-of-core. Thus, we need an algorithm for computing the dependency relation in the MT out-of-core. We present the design of such algorithm in the next Section.

6.6 Building a Multi-Tessellation out-of-core

In this section, we briefly sketch the algorithm for building dependency relation of the MT out-of-core from a sequence of simplifications provided by an out-of-core simplification algorithm.

Simplification algorithms start from a mesh at full resolution and iteratively remove details (bottom-up approach), until the resulting mesh, called the *base mesh*, is sufficiently simplified. In order to build a multiresolution model, we need to know the sequence of operations performed by the simplification process. Starting from the coarse mesh and reversing this coarsening sequence, we simulate a top-down mesh refinement. In this way we can construct the MT. The main construction steps are:

- computation of the dependency relation;
- encoding of modifications;
- storing modifications and DAG out-of-core with the selected caching policy.

This is the first module of a prototype system for out-of-core modeling that we are developing by using the MT library.

Definition of the input

Incremental simplification algorithms produce a history of the mesh updating process by generating, for each modification, the set of simplices which are removed by the modification and those which are generated by it. Actually, if the simplification algorithm is based on a decimation approach and produces a coarsening sequence, we simply reverse it to generate a refinement sequence.

Since we are dealing with huge data sets, we cannot fit the list of vertices in main memory. For this reason most out-of-core algorithms adopt a soup of top simplices in which each

simplex is defined by the coordinates of its vertices, and not an indexed data structure, where the top simplices are encoded through the indexes of the vertices.

A query on a multiresolution model should return a mesh described as an indexed data structure, or even better an indexed data structure with adjacencies. Since a soup of simplices does not maintain the connectivity of the mesh defined by the adjacencies, it should be recomputed before building the multiresolution model, or at query time.

We have decided to adopt a compact indexed encoding of modifications. Each modification $u = (u^-, u^+)$ is encoded that stores, for each modification performed by the simplification tool, the list of the vertices involved in the modification and the list of simplices with pointer to local vertices. For each vertex we store its coordinates and its global index.

Computation of the dependency relation

To compute the dependency relation, we need to perform the following operations:

- given a modification $u = (u^-, u^+)$, retrieve the top simplices belonging to u^- in the current mesh;
- since each top simplex σ in the current mesh is associated with the index of the modification that has introduced σ, identify the set of modifications $\{u_1, \dots u_p\}$ from which u depends;
- add modification u in the list of children of modifications $u_1, \dots u_p$.

The current mesh must be stored in secondary memory. To this purpose we use a PK PR k-d tree (described in Subsection 6.3.5). Recall that, at the beginning, the current mesh is the coarsest mesh produced by the simplification tool while at each step of the building process, the current mesh is refined by applying one modification. For each top simplex σ, we store the index of σ (computed during the building process) and the index of the modification that introduced σ. As search key in the PK PR k-d tree we use the centroid of σ.

We also need to store in external memory the sequence of modifications in the MT. For each modification $u = (u^-, u^+)$ we store:

- the list of the parents of u: modification from which u depends;
- the list of the children of u: modification depending on u;
- the indexes of the top simplices in u^-;

- the indexes of the top simplices in u^+ and, for each of them, the indexes of their vertices;
- number of simplices belonging to u^+ still present in the current mesh (useful to identify fully visited nodes);
- the coordinates, attributes and indexes of new vertices (if any) introduced by the modification;
- the approximation error associated with σ.

We need to store, on each disk page, a set of k modifications, in the order induced by the simplification process. In this way, we can perform an out-of-core selective refinement query that give as result the desired sorting. Since each page contains a fixed number of modifications we do not need to maintain an index, and the modification of index i_m can be retrieved with an I/O operation in the worst case. It is stored in disk page number i_m/k and the offset within the disk page is computed with the modulo-k operator. Even in this case we need to adopt a cache, in order to reduce I/O operations.

Building of the OOC MT

To build an optimized out-of-core MT we need to compute the clusters, i.e. which modifications are stored in the same disk page and to encode modifications and dependency relation. The most interesting part is the computation of clusters. Starting from the data structure adopted for computing the dependency relation, we need to perform a `GrB` sorting and encode in the final OOC MT, modifications of the MT in the order of the traversal. It consists in a breadth-first traversal of the set of modifications, with inclusion of ancestors, through an invocation of procedure FORCE_REFINE. It is similar to the top-down selective refinement, but it is simpler since we do not have to check a user-specified LOD criterion, but we need to traverse all modifications.

Chapter 7

Towards a multiresolution structural representation for a scalar field

7.1 Introduction

The purpose of this work, as started in the introduction to this thesis, has been to enhance the expressive power of multiresolution models for scalar fields by including a morphological description of the field. Current multiresolution models for free-form surfaces, and scalar fields use a geometry-based representation of the surface, or field, as a simplicial mesh, but they do not encompass a structural, or morphological, description. A morphological description of a scalar field, defined by the Morse-Smale complex, also provides a concise representation of the field which can be alternative to a mesh based one. On the other hand, the morphological description can be combined with in a geometry-based representation.

In this Chapter, we present some preliminary results in this direction. We have focused on encompassing a morphological representation of a scalar field in a multi-resolution model, and have considered the case of two-dimensional scalar fields. Our long term goal is to develop morphology-driven multiresolution models of 3D scalar fields as well as hierarchical representations of scalar fields based on morphology.

We have developed heuristics for computing an approximated Morse-Smale complex from a triangle-based terrain model [DFMD02, DDFM$^+$03a]. Unlike the algorithms described in Section 3.5, which compute the boundary of the regions in a Morse-Smale complex described by the critical net, we have developed a region-based approach, which compute the regions of the stable and unstable decomposition as collections of triangles. This approach can be easily extended to the d-dimensional case, and actually we have performed a dimension-independent implementation of one of the two algorithms we describe here.

Our other contribution has been to design and develop a simplification algorithm which maintains the topology of the Morse-Smale complex at different resolution, as a first step in the direction of a morphology-based multiresolution model of a scalar field [DDFM$^+$03a].

7.2 Morse-Smale complex in the discrete case

We consider the problem of defining and extracting an approximation of the Morse Smale complex for a two-dimensional scalar field described by a *Triangulated Irregular Network (TIN)*. Thus, we consider a C^0 approximation of the field based on a triangle mesh. The approximation of the Morse-Smale complex should be a decomposition of the TIN into 2-cells that correspond to the 2-cells of the Morse Smale complex in the C^2-differentiable case. To this aim, we introduce a region-based constructive definition of the Morse- Smale complex in the discrete case.

As in the C^2 differentiable case, the Morse-Smale complex in the discrete case is the intersection of the stable and the unstable cell complex. We define the 2-cells of the stable and unstable complexes in a constructive way by using a region-growing approach. We call the resulting complexes approximated *stable* and *unstable complexes*, respectively.

Each 2-cell in the approximated stable, or unstable, complex is a 1-connected set of triangles of the TIN. A 2-cell γ in the approximated stable complex corresponds to a relative minimum m_R and has the property that, for each point p in γ, there exists a directed path P joining m_R to p which belongs to γ in which the orientation of the gradient on the triangles intersected by P is always opposite to the orientation of the path. Similarly, a 2-cell γ of the approximated unstable complex corresponds to a relative maximum M_R such that the exists at least one directed path P joining M_R to any point p of γ such that the gradient of each intersected triangle has the same orientation as the path. The intersection of the approximated unstable and stable complex gives the *approximated Morse-Smale complex*. The boundaries of the 2-cells of the approximated Morse-Smale complex are chains of edges of the TIN, which form an approximation of the critical net. We call the 2-cells in the stable and unstable regions.

Note that the above definition can be naturally extended to the d-dimensional case, with $d > 2$, i.e., to d-dimensional scalar fields. In this case, each d-cell of the approximated Morse-Smale complex would be a $(d-1)$-connected set of d-simplices. The $(d-1)$-boundaries of such cells will form the generalization of the critical net in the d-dimensional case.

7.2.1 Constructing a region in an approximated unstable Morse-Smale complex

In this subsection, we describe an algorithm for computing the 2-cells (regions) of the approximated unstable complex through a region-growing approach. We denote with R_p the region of the unstable complex generated by a relative maximum p. The stable complex can be built in a similar way. We denote as B_p the boundary of R_p. We describe how to construct a region R_p independently of the criterion used to extend a 2-cell.

Initially, we sort all vertices of the TIN in decreasing order, based on their field value. The vertex p with highest field value is selected. Point p is marked as a critical point (relative maximum). In this first iteration, p is also the global maximum. The region R_p corresponding to p is initialized with the triangles incident in p. R_p is extended by adding a new triangle t at a time, selected in such way that triangle t has an edge on B_p. The criterion to select a new triangle depends on the specific heuristic algorithm used to simulate a region of an unstable Morse-Smale complex.

A region is extended as long as we find adjacent triangles that can be added. When the construction of region R_p is completed, we extract from the sorted list the next vertex q having the maximum field value, and not belonging to the interior of one of the unstable regions already computed. In a generic expansion step, primitive ADD_TRIANGLE(t, e, R_p) is invoked to visit a triangle t starting from one edge e of t belonging to the boundary B_p of region R_p. Such primitive returns triangle t', adjacent to t and incident in e, it if can be added to the region R_p.

When a vertex q is considered for generating a new region R_q, we test whether q is a local maximum. If not, then some of the triangles incident in q must have been visited before. Hence, q lies on the (current) boundary of some region(s) already computed. In this case, we merge region R_q into one of those regions. We select the one corresponding to the maximum with the highest elevation. The algorithm stops when all triangles have been assigned to some region.

For each vertex p of the TIN, the algorithm uses a list L_p which stores the set of triangles incident in p and not yet inserted in any unstable region. When a triangle t is inserted into a region, procedure REMOVE(t) retrieves its vertices p_1, p_2 and p_3, deletes triangle t from lists L_{p_1}, L_{p_2} and L_{p_1} and, if one of the lists becomes empty, the corresponding vertex is marked as visited. In this case, vertex p belongs either to the interior of a region, or to the boundary of two or more regions that have been completely visited. Such vertex p cannot be a relative maximum of the remaining mesh, and we will skip it in the sorted list of vertices. A pseudo-code description of procedure REMOVE is reported below.

```
Procedure REMOVE(triangle t)
/* remove t from each list L_p, with p vertex of triangle t */
   for each p vertex of t do
      L_p.del_item(t)
/* if list L_p is empty, all triangles incident into p are visited */
      if L_p = ∅ then
         p.visited = true
   endfor
end remove
```

The expansion of the current unstable region R_p is performed through a breadth-first traversal algorithm. We insert the boundary edges of R_p, along which R_p could be expanded, into a queue Q. Such queue is dynamically updated during the algorithm. At each step, procedure EXPAND_REGION extracts an edge e from Q, invoke primitive ADD_TRIANGLE that checks triangle t', which is adjacent to edge e. Then t' is skipped or added to the region, according to ADD_TRIANGLE. If t' becomes part of the region, one or more of its edges are added to queue Q, according to primitive ADD_TO_QUEUE. When the queue is empty, the region cannot be expanded further. A pseudo-code description of procedure EXPAND_REGION is reported below.

```
Procedure EXPAND_REGION(vertices V, triangles T, queue Q, region R)
/* expand region R along edges in Q */
   while not is_empty(Q) do
      Q.del_item(< t, e >)
/* check if a neighbor of t along e can be added to R */
      if ADD_TRIANGLE(t, e, R)=t' then
         R = R ∪ t'
         REMOVE(t')
         ADD_TO_QUEUE(Q, t', e, R)
      endif
   endwhile
end EXPAND_REGION
```

Our implementation of the decomposition algorithm is based on encoding the TIN as an extended indexed data structure with adjacencies (see Subsection 2.2.2). For each vertex p, we store its coordinates, the field value and the triangles in the star of p. For each triangle t, we store the indexes of its vertices, the triangles adjacent to it along its edges. The above process is described by the following pseudo-code.

```
Procedure COMPUTE_UNSTABLE(vertices V, triangles T)
   Q = empty_queue()
/* sort the vertices V of the TIN by decreasing field value */
   Sort(V)
   for each p in V do
      p.visited = false
      L_p = set of triangles in the star of p
   endfor
   while V ≠ ∅ do
      /* consider the first non-visited element of V */
      do p = V.first() until p.visited = false
      if no triangle incident in p is visited then
/* p is a relative maximum, generate new region R_p */
/* insert in the new region R_p the star of p */
         R_p = L_p
         for each t in L_p do
            REMOVE(t)
/* insert in the queue Q the link of p */
            Q.add_item(< t, e >, R_p)
         endfor
         EXPAND_REGION(V, T, Q, R_p)
      else /* p is on the boundary of existing region R_q */
/* add to R_q the triangles incident in p */
         R_q = R_q ∪ L_p
         for each t in L_p do
            REMOVE(t)
/* insert in the queue Q the link of p */
            Q.add_item(< t, e >, R_q)
         endfor
         EXPAND_REGION(V, T, Q, R_q)
      endif
   endwhile
end COMPUTE_UNSTABLE
```

The algorithm for computing the unstable decomposition finds just the relative maxima, as the points of maximum field value not lying on the boundary of any region. Similarly, the algorithm for computing the stable decomposition finds just the relative minima. Therefore, the algorithm for computing the Morse-Smale decomposition finds both maxima and minima. It detects saddle points as those points that belong to the boundary of both an unstable and a stable region.

7.3 A discrete technique

The technique we proposed in [DFMD02] for extending a region in the approximated unstable complex, uses a heuristics which is simple to implement, fast to execute and easily to extend to arbitrary dimensions. Our implementation can also handle arbitrarily-dimensional simplicial complexes.

As in [TIKU95], such technique computes the slope of a triangle based just on field values. A triangle t having vertices p_1, p_2 and p_3, external to a region R and incident into its boundary B along edge $e_1 = (p_1, p_2)$, is inserted into region R if the other vertex p_3 of t, not belonging to the boundary of R, has a field value less than p_1 and p_2. Intuitively, this condition tests the matching between the triangle gradient and the out coming vector from the middle point of e_1 directed towards p_3. In the following, we describe the implementation of the primitives that, together with the algorithm for visiting the regions, described in Subsection 7.2.1, completely define the technique for computing the unstable regions.

Given a triangle t belonging to the region R and one edge e of t, primitive ADD_TRIANGLE checks whether triangle t', opposite to t with respect to e, exists, it does not belong to any other region, and the field value at the vertex of t' opposite to e is lower than the field value at the two vertices of e. In positive case, it returns t', thus enabling adding it to region R. Given a triangle t and an edge e of t, primitive ADD_TO_QUEUE enqueues the two edges of t different from e. The version of primitive ADD_TRIANGLE used in the algorithm for computing the stable decomposition is similar, but it checks that the field value of the third vertex is larger, rather than smaller, than the two vertices of e.

Our implementation of the decomposition algorithm is dimension-independent and visits each $(d-1)$-simplex exactly twice. Thus, this is linear in the number of $(d-1)$-simplexes. In the 2D case, where the number of edges is in $O(n)$, the algorithm has a $O(nlog_2 n)$ worst-case time complexity, which is dominated by the cost of sorting the vertices of the TIN according to the field value. In the d-dimensional case, the worst-case time complexity is linear in the number of d-simplexes and, for instance, it is in $O(n^2)$ in the 3D case.

7.4 A C^0 technique

In this Section, we describe a region-based technique we have proposed in [DDFM+03a] for computing an approximation of the Morse-Smale complex based on a piecewise linear approximation of the terrain dataset.

To this aim, we define the direction of maximum slope for each triangle t of the TIN, by considering the gradient of the linear function that interpolates the elevations at the three vertices of t. Using such gradient direction, we investigate how integral lines traverse a

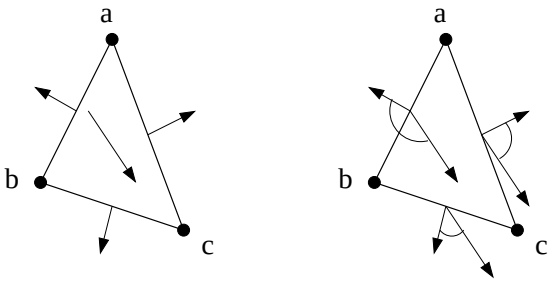


Figure 7.1: A triangle $t = abc$ with its gradient, and the normals to its three edges. Edge ab, is an entrance, while ac and bc are exits. The best exit is bc, since its normal minimizes the angle with the gradient of t i.e., maximizes the inner product.

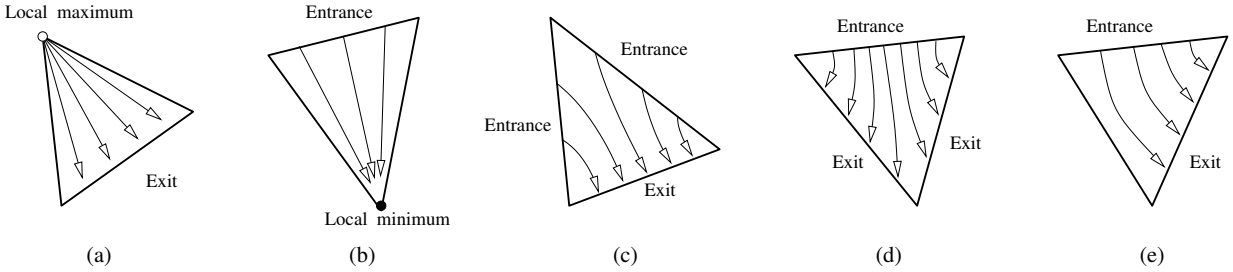


Figure 7.2: The five configurations of bundles of integral lines traversing a triangle.

triangle t.

Let e be an edge of a triangle t. We say that e is an *exit* for triangle t if the gradient of t and the unit normal of e outgoing from t form an angle $< 90^o$, i.e., their inner product is > 0. We say that e is an *entrance* for t is the gradient of t and the unit normal of e outgoing from t form an angle $> 90^o$, i.e., their inner product is < 0. Entrance and exit of a triangle are depicted in Figure 7.1.

Since integral lines follow, at each point, the direction of steepest descent, a triangle is traversed by a *bundle of integral lines,* which enter t at its entrance and leave it at its exit. If a triangle t has two entrances, then two different flows of integral lines merge at t to form its bundle. Similarly, if t has two exits, its bundle is split into two different flows when leaving t. However, there can be flow across an edge e if and only if e is an entrance for one of its incident triangles, and an exit for the other of its incident triangles. Otherwise, the flow will travel along e. Edges that cannot be traversed by the flow form natural "barriers". There are five possible configurations of bundles traversing a triangle (see Figure 7.2), and two possible configurations of bundles either converging at, or emanating from an edge (see Figure 7.3).

A first attempt to define an approximated Morse-Smale decomposition would be to form

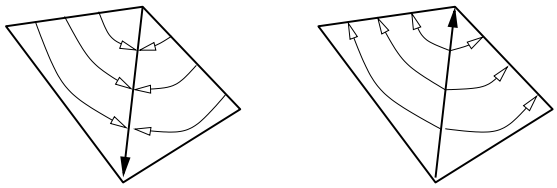


Figure 7.3: The two configurations of bundles of integral lines traveling along an edge: a a valley, and b a ridge.

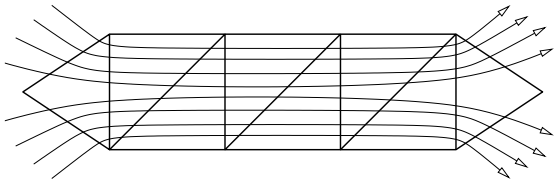


Figure 7.4: Two separate flows of integral lines merging at the same bundle and splitting later on.

components by clustering those triangles of the TIN that contain bundles emanating from a common local maximum. Referring to Figure 7.2, triangles forming a region would be found by traversing the TIN, starting at triangles with configuration (a), and following bundles through adjacencies. However, this definition has problems, because triangles with configuration (c) may merge bundles originating from different local maxima, which will travel in parallel for a while, and eventually split through a triangle with configuration (d), as depicted in Figure 7.4. A simple topological traversal of the TIN is, thus, not sufficient to discriminate between triangles belonging to different components. The separatrix between the components of such two local maxima should actually split triangles traversed by those "mixed" bundles. On the other hand, our aim is to form components by clustering triangles, and, thus, we want to avoid splitting a triangle between two different components. We will rather use a heuristic approach that constrains all separatrices to lie at edges of the TIN. In a case like the one depicted in Figure 7.4, all triangles in the strip will be assigned to one of the two local maxima originating the bundle, and, thus, their separatrix would follow the border of the strip.

In order to implement the new algorithm, we added a preprocessing step to the algorithm described in Section 7.3. In the preprocessing step, we first compute the gradient for each triangle t of the TIN. Then, we identify regions composed of horizontal triangles, i.e., triangles with null gradient. Such regions are marked as plateaus and will not be considered in the next stages of the algorithm. This operation is necessary because real terrains are, in general, not Morse functions. Finally, for each non-plateau triangle t, we compute the inner product between the gradient of t and the normal to each of its edges, in order to

identify the entrance s and exit s of t. In case a triangle t has more than one exit, we also define its *best exit* as the one having its normal better aligned with the direction of steepest descent. This corresponds to the minimum angle, i.e., the maximum positive value of the inner product between the gradient of t and the normal to the edge, as illustrated in Figure 7.1.

Initially, all triangles except from plateaus are unmarked. At each iteration of the main loop, the current vertex p is the one having the maximum elevation among the vertices of unmarked triangles. Each region R_p is initialized by all unmarked triangles incident at p. Then an inner loop starts, which performs region growing by following bundles of integral lines through adjacent triangles. With respect to the region-growing algorithm in Section 7.3 primitives ADD_TRIANGLE and ADD_TO_QUEUE have been modified.

For each such pair $< t, e >$, which belongs to the queue Q, ADD_TRIANGLE() expands R_p to include its neighbor t' along its best exit e if and only if t' exists, t' was not visited yet, and e is an entrance of t'. In this case triangle t' is also marked as visited. ADD_TO_QUEUE() inserts in the queue those pairs (t', e'), where edge e' is the best exit for the triangle t' and the edge is part of the boundary of R_p.

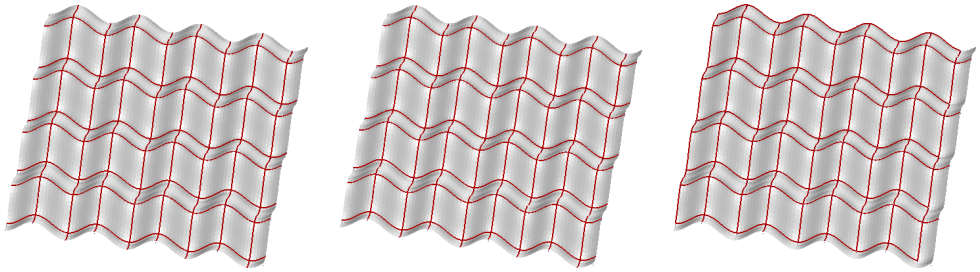
Note that we do not grow across both exits of triangles in configuration (d), but we only use the best exit, as in configuration (e). This is a heuristic criterion adopted in order to avoid an overflow of the component under construction into a neighboring component, which could occur in situations like the one in Figure 7.4, if both exits were used. On the other hand, the region of a given local maximum p will be not necessarily completed in a single iteration starting at p. This region will be completed later on, when visiting vertices on the boundary of the current region R_p.

7.5 Analysis and comparison

7.5.1 Experimental results

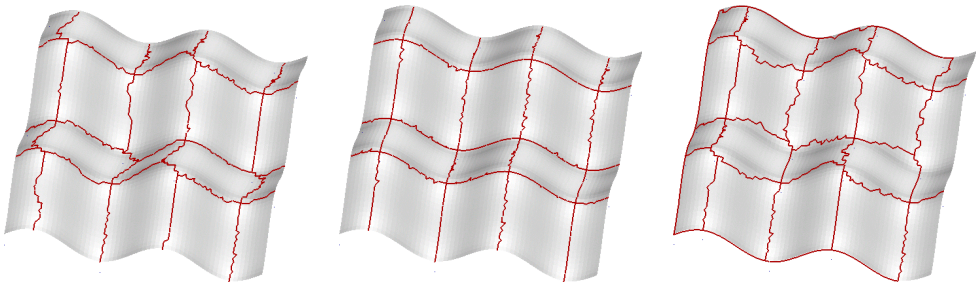
In this Section, we compare our techniques, and we show results produced by the algorithm which computes an approximated Morse-Smale complex based on the C^0-approximation. We present experiments on both synthetic and real datasets.

On a simple regular dataset composed by 6561 vertices and 12800 triangles representing the function $f(x, y) = \sin(x) + \sin(y)$ both our techniques perform well. See Figure 7.5 (a) for the discrete technique and Figure 7.5 (b) for the C^0 technique. On a dataset with 10658 vertices and 20940 triangles, representing the same function, through regular samples plus points on isolines, results are quite different. Figure 7.6 (a) for the results with the discrete technique and Figure 7.6 (b) presents the results obtained with the C^0 technique, which



(a) **(b)** **(c)**

Figure 7.5: Approximated Morse-Smale complex for the regular dataset composed of 6561 vertices and 12800 triangles representing function $f(x, y) = \sin(x) + \sin(y)$. Results with (a) discrete, (b) C^0 and (c) Takahashi's approaches are shown.



(a) **(b)** **(c)**

Figure 7.6: Approximated Morse-Smale complex for the irregular dataset composed by 10658 vertices and 20940 triangles representing function $f(x, y) = \sin(x) + \sin(y)$. Results with (a) discrete, (b) C^0 and (c) Takahashi's approaches are shown.

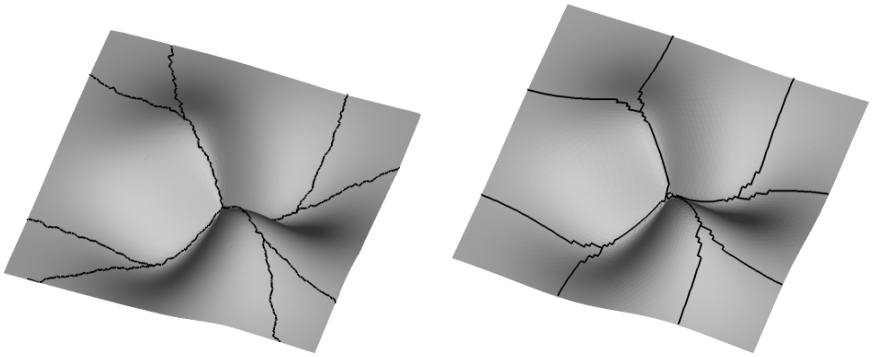
is clearly better. For comparison in Figures 7.5 (c) and 7.6 (c) we show the critical net computed by Takahashi's approach [TIKU95].

Synthetic data are obtained by sampling function $f(x, y) = x \cdot y \cdot e^{-(x^2+y^2)}$ on a square of side length 2 centered at the origin. Figures 7.7 (a) and 7.7 (b) refer to random sample (32,000 vertices, 63,971 triangles), and regular sample a grid (6,561 vertices,12,800 triangles), respectively. Function f is a Morse function having two peaks at points $(\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}})$ and $(\frac{-1}{\sqrt{2}}, \frac{-1}{\sqrt{2}})$, two pits at points $(\frac{1}{\sqrt{2}}, \frac{-1}{\sqrt{2}})$ and $(\frac{1}{\sqrt{2}}, \frac{-1}{\sqrt{2}})$ and one saddle at the origin. Moreover, there are two peaks and two pits at the corners of the square boundary.

An analytic study of f gives

$$Grad(f) = \left((-2x^2 + 1)ye^{-(x^2+y^2)}; (-2y^2 + 1)xe^{-(x^2+y^2)}\right).$$

Therefore, critical points are $p_0 = (0, 0)$, $p_1 = (\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}})$, $p_2 = (\frac{-1}{\sqrt{2}}, \frac{-1}{\sqrt{2}})$, $p_3 = (\frac{1}{\sqrt{2}}, \frac{-1}{\sqrt{2}})$, $p_4 = (\frac{1}{\sqrt{2}}, \frac{-1}{\sqrt{2}})$. The Hessian matrix of each critical point is equal to



(a)
Random points
(32000 vertices, 63971 triangles)

(b)
Points on a grid
(6561 vertices, 12800 triangles)

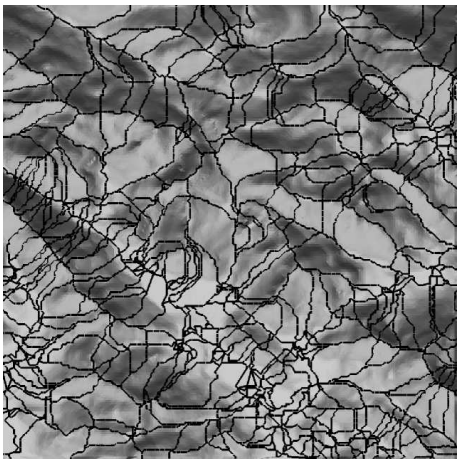
Figure 7.7: Approximated Morse-Smale complex for the synthetic function modeled by two different TINs.

$$Hess_0 f = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$
$$Hess_{p_1} f = Hess_{p_2} f = \begin{pmatrix} -2 & 0 \\ 0 & -2 \end{pmatrix}$$
$$Hess_{p_3} f = Hess_{p_4} f = \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix}$$

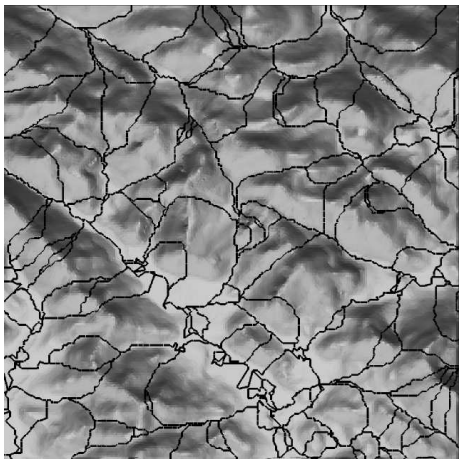
Since the determinant of each Hessian matrix is different from 0, then function f is a Morse function. The determinant of $Hess_O f$ is negative, then p_0 is saddle point. The eigenvalues of $Hess_{p_1} f$ and $Hessp_2 f$ are all negative (equal to -2), then points p_1 and p_2 are maxima. Finally, points p_3 and p_4 are minima since the eigenvalues corresponding to their Hessian matrix are positive (equal to 2).

The exact critical net of f in the square is made by four integral curves connecting the origin to the other critical points. In addition, our algorithm generates curves connecting minima/maxima to the boundary of the square, because it takes into account the maxima and minima lying on the boundary. These points are not considered in the smooth case, since the gradient field and Hessian matrix are not well defined at boundary points.

In both the random sampled and regular datasets, the regions generated by our algorithm match the actual surface features. Some portions of critical lines are jagged because edges of the TIN are not aligned with critical lines. Some geometrical difference exists between the results obtained from the random and from the griddled input. Note that, in the latter case, a (false) tiny plateau is detected at the saddle. Such effects are due to discretization, and do not affect the validity of our approach. Figure 7.8 shows the results obtained for real data representing a terrain in the area of Mount Marcy, NY (courtesy of the U.S.



Extended stable and unstable decompositions



Extended unstable decomposition

Figure 7.8: Results for Mount Marcy data set. The TIN has 35162 scattered vertices, 69718 triangles.

Geological Survey). Underlying data have an irregular distribution (obtained by selective subsample from a grid). The TIN consists of 35,162 vertices and 69,718 triangles. Results appear to be consistent with the real terrain morphology.

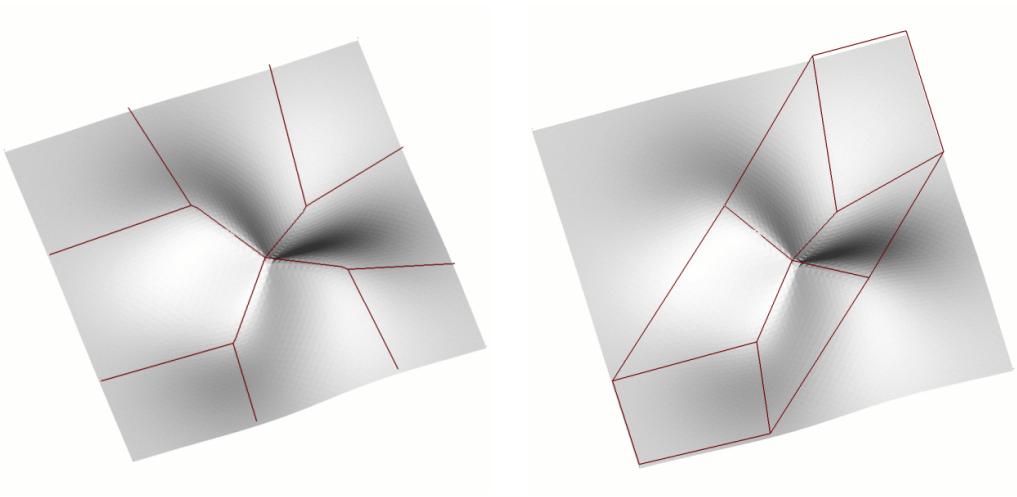
7.5.2 Comparisons

We have compared the technique described in Section 7.4 with the method based on direct computation of the critical nets discussed in Section 3.5. In particular, we have implemented the method proposed in [TIKU95].

The method in [TIKU95] extract first the critical points by using the technique described in Section 3.5. Our method computes maxima and minima when building the stable and unstable decomposition. Saddle points are intersection of the boundaries of the stable and unstable Morse-Smale complexes. Our experiments have found that both algorithms extract all critical points.

Essentially, differences appear on the boundary of the dataset. The method in [TIKU95] consider the TIN as a mesh homeomorphic to a sphere, by adding a virtual point, called the *virtual pit*, of absolute minimum. Therefore, each point on the boundary of the mesh is adjacent to the virtual pit. This implies that it is not possible to recognize minima lying on the boundary of the domain, since all boundary vertices are adjacent to the virtual pit, which is the absolute minimum.

The introduction of the virtual pit also affects the recognition of saddle points lying on



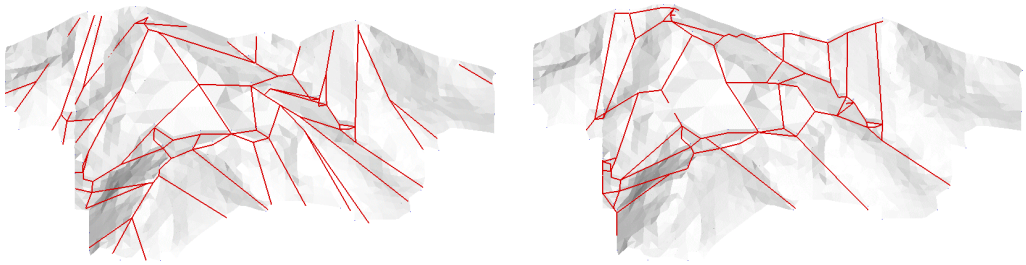
(a) **(b)**

Figure 7.9: Comparison among surface net, on a synthetic dataset, extracted by technique described in section 7.4, shown on the left, and Takahashi's approach [TIKU95], shown on the right.

the boundary. Let us consider a valley having its maximum on the boundary. According to the method [TIKU95], such valley has a saddle at that point, instead of a maximum, while a ridge having its minimum on the mesh boundary does not give raise to a saddle point. Since the two configurations are symmetric, it would be preferable to handle them in a uniform way. Our technique recognizes both cases as saddle points. Figure 7.9 (a) shows the surface net extracted with the technique presented in Section 7.4, while Figure 7.9 (a) shows the surface net extracted with Takahashi's technique [TIKU95]. Note that with Takahashi technique four saddle points are missing, and the quality of the resulting surface net is compromised.

As a consequence of the above, the technique in [TIKU95] is not invariant with respect to the sign of the field value. Let f and g be two fields such that $f(p) = -g(p) = -z$ for each vertex p of the TIN. According to [TIKU95], there are points q that are maxima for f and not minima for g, and saddle points for f that are not saddle points for g, or vice versa.

Even in cases in which the critical points found by the two techniques are the same, there may be substantial differences in the critical net. This is due to the different basic approach. Takahashi's approach builds paths starting just from the saddle points, but saddle points are not always present in the mesh. If our data set represents a part of a mountain, saddle point(s) may be outside the sampled area, and thus the algorithm does not find any edge in the critical net. On the contrary, our technique starts from relative maxima and minima, and in a closed domain such points are always present. Although, in practice, TIN representing terrains completely without saddle points are extremely rare,



(a) **(b)**

Figure 7.10: Comparison on surface network on a real dataset composed of 6095 vertices and 11838 triangles from Italy: (a) our technique and (b) Takahashi's approach.

the case of valleys, having their saddle points outside the mesh, is more frequent. Such portions of the critical net have not been found by Takahashi's technique. On the contrary, they have been correctly found by our technique (see Figure 7.10 and 7.11).

7.6 Multiresolution TIN

In this Section, we describe a technique for maintaining the morphology of a terrain inside the multiresolution model. To this aim, we build a Multi-Triangulation (MT), starting from the input TIN and its associated Morse-Smale complex, through a sequence of coarsening modifications. In this way, we can extract TINs from the MT, in which the original terrain morphology is faithfully represented.

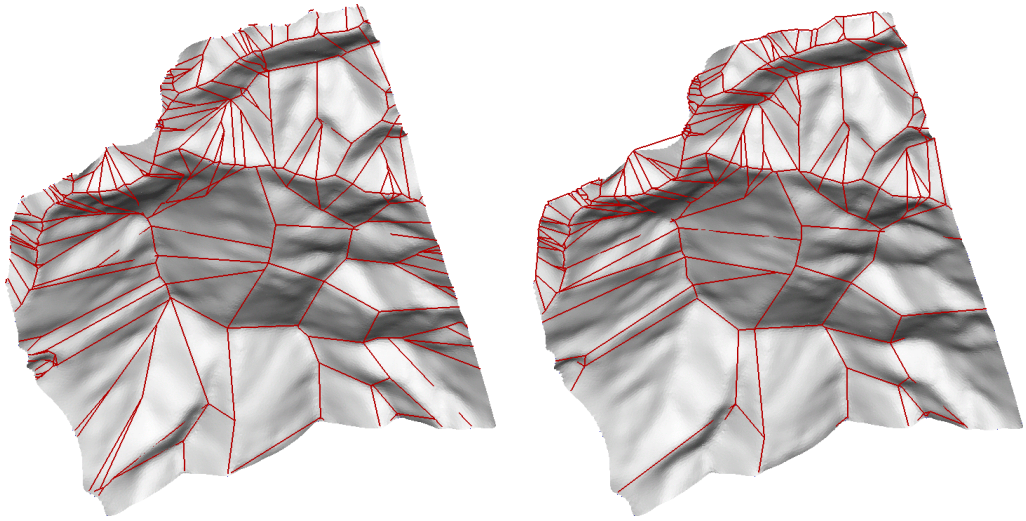
7.6.1 Simplification Algorithm

The simplification algorithm consists of a vertex decimation in a constrained triangle mesh. We consider the TIN as a constrained triangulation where the constraints are the edges of the critical net. For simplicity, we also define the edges lying on the boundary of the domain to be constraints.

The algorithm is iterative. At each step, it removes one vertex. A vertex v is removable if one of the following two conditions holds:

(i) no constraint edge is incident at v;

(ii) exactly two constraint edges are incident at v, and, if v is on the boundary of the domain, such two edges are aligned.



(a) **(b)**

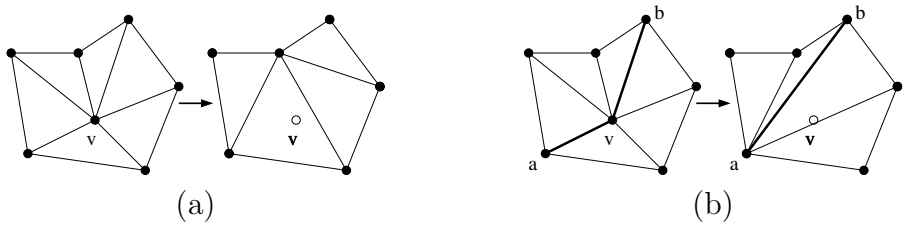
Figure 7.11: Comparison on surface network on a real dataset composed by 46765 vertices and 92400 triangles from New Zealand: (a) our technique and (b) Takahashi's approach.

Vertices having three or more constraint edges incident at them cannot be removed. This implies that saddle points will never be removed, as they have at least four incident constraint edges. Therefore, vertex removal does not alter the topology of the critical net and the morphology of the isolines. The last clause in condition (ii) also implies that the shape of the domain will not be modified.

When a vertex v is removed, all triangles incident at it are removed. Such triangles form the set u^- of the coarsening modification $u = (u^-, u^+)$ that we are applying. The deletion of set u^- leaves a polygonal hole π in the triangulation. If v has no incident constraints, then π is re-triangulated according to the Delaunay criterion (see Figure 7.12 (a)). If v has two incident constraints, av and bv, respectively, then π is triangulated through a Delaunay triangulation constrained by edge ab (see Figure 7.12 (b)). The new triangulation of π forms u^+.

In simplification, at each step, we remove the vertex causing the least increase in the approximation error, among the set of removable vertices. An approximation error is defined for each triangle t of the TIN as the maximum vertical distance from t of data points which project inside or on the boundary of t. The approximation error of the TIN is the maximum error of all its triangles.

At each step, the error that would be introduced by removing a vertex v can be evaluated in the following way. We simulate removing v, we compute the new triangles and the approximation error of such triangles, and we take the maximum of such errors. Since this approach is quite expensive, we estimate the error caused by removing v in an approximated



(a) (b)

Figure 7.12: (a) Modification when removing a vertex with no incident constraints, and (b) modification when removing a vertex with two incident constraints (constraint edges are thick).

way, i.e., by taking the vertical distance of v from the horizontal plane lying at the average elevation of the vertices adjacent to v.

The algorithm stops when either a user-defined maximum admissible approximation error has been reached, or a user-defined number of vertices has been removed, or when there are no more removable vertices.

7.6.2 MT Construction

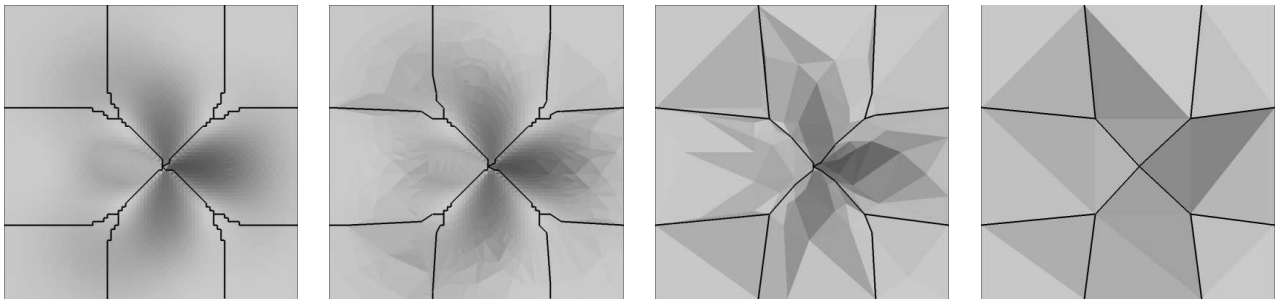
Since the simplification algorithm is based on vertex decimation, the (coarsening) modifications performed during this algorithm are vertex removals. Thus, the corresponding refinement modifications in the resulting MT will be vertex insertions. The MT is built by setting its base TIN as the simplified TIN output by the decimation algorithm, and its set of modifications as the set of inverse modifications applied during decimation. The partial order among modifications is set according to the definition given in Section 4.1.1.

7.6.3 Experimental results

We have performed experiments on MT built as described above. We have extracted from such MTs a set of TINs at decreasing levels of detail, by using threshold conditions based on a sequence of increasing error values.

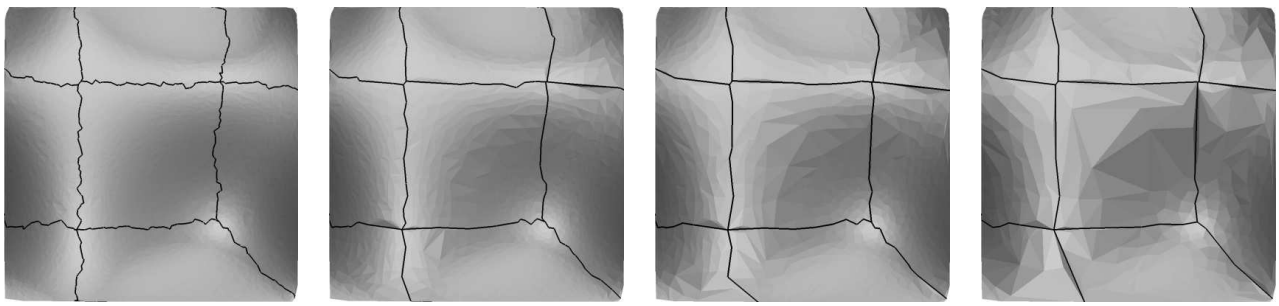
Figure 7.13 shows an example of the sequence of extracted TINs for the synthetic function introduced in Subsection 7.5.1 and for the MT built from gridded sample data. It is easy to verify that the geometry of the Morse-Smale complex is progressively simplified while reducing the LOD (i.e., increasing the error), but its connectivity structure is preserved.

Figure 7.14 an example of the sequence of extracted TINs for the synthetic function $(x^3 - 3x+3)\cdot(y^3-3y+3)$ randomly sampled. Also in this case the geometry of the Morse-Smale complex is progressively simplified while reducing the LOD (i.e., increasing the error), but



12289 triangles, 305 critical edges | 2212 triangles, 150 critical edges | 141 triangles, 23 critical edges | 20 triangles, 12 critical edges

Figure 7.13: Morse-Smale complexes for some TINs extracted from the MT built for the synthetic function (points lying on a grid).



14326 triangles, 337 critical edges | 3768 triangles, 103 critical edges | 1850 triangles, 62 critical edges | 816 triangles, 39 critical edges

Figure 7.14: Morse-Smale complexes for some TINs extracted from the MT built for a synthetic function randomly sampled.

its connectivity structure is preserved.

7.7 Conclusions

In this chapter we have presented two heuristic algorithms to construct a Morse-Smale decomposition in the discrete case [DFMD02, DDFM$^+$03a]. One of them is designed and implemented in arbitrary dimension, while the second is for triangle-based terrain models, but it can be extended to simplicial models in arbitrary dimension.

We have analyzed such techniques on both real and on synthetic datasets, and compared with a technique that computes the boundary of the regions in Morse-Smale complex provided by the critical net [TIKU95].

We have also proposed a technique for simplification and multiresolution modeling of a terrain, that extracts and maintains the morphological structure of the terrain, at different levels of detail. Terrain representations at different LODs, extracted from such model, maintain the topology of the isolines of the TIN at full resolution. Computing the domain decomposition, and simplifying the TIN by preserving the critical net topology, introduces a negligible overhead with respect to computing a multiresolution TIN without taking topology into account.

The results presented in this chapter are just a first step in the direction of morphology-based scalar field representation. Current and future developments of the work described in this chapter include:

1. experimental comparisons with watershed techniques in the case of 2D scalar fields;
2. extension of the C^0 method to the case of 3D scalar fields
3. development of a watershed-based approach in 3D and comparison with the approach in 2;
4. development of a multiresolution model based on morphology for 3D scalar fields.

Chapter 8

Conclusion

Contribution of this thesis is in the field of efficient modeling and storing of large two and three-dimensional discrete scalar fields through a multi-resolution (also called Level-Of-Detail (LOD)) approach. LOD models are a powerful representation since a user can dynamically and efficiently select a subset of the high resolution model. On the other hand, multi-resolution data structures which explicitly encode the modifications in an LOD model have large space requirements, and often cannot fit in main memory.

This problem can be avoided with compact multi-resolution data structures or with out-of-core multi-resolution modeling. In this thesis, we have first developed and experimented compact data structures for multi-resolution models for two-dimensional scalar fields based on triangle meshes and built through iterative vertex-insertion. Such representations can be also applied to encode multi-resolution models of free-form triangulated surfaces. We have then considered the problem of developing efficient data structures as well as construction and query algorithms for multi-resolution models for three dimensional scalar fields based on unstructured tetrahedral meshes. Specifically, we have designed and developed three multi-resolution data structures, based on vertex insertion/removal, the *Vertex-based 3D MT* and on half-edge collapse, the *Half-Edge MT* and the *Half-Edge Tree (HET)*. We have compared such representations with an LOD tetrahedral model based on full-edge collapse and with a regular LOD model based on nested tetrahedral meshes.

Largest datasets exceed main memory even if compact data structures are adopted. For these reasons, we have investigated out-of core approaches for encoding and manipulating multi-resolution models for 3D shapes, two- and three-dimensional scalar fields, independently of the simplification strategy used to generate the LOD model. The major issue here has been investigating and experimenting effective techniques for clustering the updates in the multi-resolution model. In our work, we have developed a large variety of clustering techniques for an out-of-core multi-resolution models, and we have experimented them in

connection with several selective refinement queries.

The clustering algorithms that we have developed can be coupled with a compact or with an explicit representation of the model, and are the basis for the development of a system for out-of-core multi-resolution modeling which can be combined with different encodings of the updates in the model. Further developments of the work described in this thesis are the implementation and experimentation of an out-of-core system for multi-resolution modeling based on the MT library and thus on an explicit representation of the MT. We plan to experiment the system on both terrain and volume data sets. For terrain data, we plan to use and modify the out-of-core simplification tool in [CMRS03a] for generating a sequence of modifications of the full- resolution mesh in secondary memory. For volume data, we are currently developing a tool to simplify tetrahedral meshes based on half-edge collapse out-of-core. At a later stage, we will also develop specific implementations of the out-of-core modeling system based on implicit representations of the modifications, in particular by using implicit procedural encodings of half-edge collapse on triangle [Paj01] and tetrahedral meshes (see Section 5.2).

Another relevant related issue is the possibility to transmit LOD models over networks with limited bandwidth and to visualize them on devices with small throughput, such ad PDAs. Specific data structures must be devised, which support client/server mechanisms and can minimize the traffic over the transmission channel. Some proposals of client/server architectures for LOD models exist in the literature [DFMMP00, ESS03, KLK04], for specific LOD data structure, and also some preliminary work has been done on applications to PDAs [LGCR04] based on the MT library for triangulated surfaced. Important yet almost unexplored issues concern cache and look-ahead policies that should further decrease the traffic over a limited bandwidth. The clustering techniques investigated in this thesis for out-of-core multi-resolution modeling can be a suitable basis for developing a communication protocol that prefetchs and groups modifications, this can reduce the latency due to the remote communication.

One of the major motivations in using multi-resolution models based on unstructured meshes for describing a scalar field lies in their ability of encompassing morphological information. This is the first step towards the development of more powerful multi-resolution models based on the combined representation of the geometry and the morphology of the field. To this aim, in this thesis, we have developed two techniques for extracting morphological information from a scalar field, and we have developed a multi-resolution model for a 2D scalar field which maintains the morphology of the field at different levels of resolution.

Our objective is designing and developing LOD approaches, capable of capturing and representing the morphological structure of a field at a range of different levels of abstraction, and which can support a hierarchical approach to analysis, inspection and understanding. We would like to encompass morphological information in a mesh-based LOD model to

provide flexible and more accurate reconstructions of the field at different resolutions. It is also interesting to investigate purely structural LOD models from which variable-resolution morphological representations can be extracted. These models can be the basis for developing automatic tools for extracting knowledge from a scientific data set. The case of volume data is almost entirely unexplored. Thus, the first step in the direction of multi- resolution modeling for 3D scalar fields should be developing techniques for extracting morphological information from such fields To this aim, we plan to investigate extensions of the method presented in Section 7.4 as well of watershed techniques, which have been developed for 2D images, to extract approximations of a Morse-Smale complex from tetrahedral volume data sets.

Acknowledgments

The work has been partially supported by:

- Project funded by the Italian Ministry of Education, University, and Research (MIUR) on *Representation and processing of spatial data in Geographic Information Systems* (SPADA).
- Projects funded by the Italian Ministry of Education, University, and Research (MIUR) *Algorithmic and Computational Methods for Geometric Object Representation* (MACROGeo).
- Project funded by the Italian Ministry of Education, University, and Research (MIUR) on *Representation and management of spatial data in the WEB* (SPADA-WEB).
- European Network of Excellence AIM@SHAPE under contract number 506766.
- University of Genova travel grant for supporting visits at the University of Maryland.
- Research funds from UMIACS, University of Maryland, (research group by Prof. Hanan Samet).

Bibliography

[AAM+98] P. K. Agarwal, L. Arge, T. M. Murali, K. R. Varadarajan, and J. S. Vitter. I/o-efficient algorithms for contour-line extraction and planar graph blocking. In *Proceedings of the ninth annual ACM-SIAM Symposium on Discrete algorithms*, pages 117–126. Society for Industrial and Applied Mathematics, 1998.

[ACW+99] D. Aliaga, J. Cohen, A. Wilson, E. Baker, H. Zhang, C. Erikson, K. Hoff, T. Hudson, W. Stürzlinger, R. Bastos, M. Whitton, F. Brooks, and D. Manocha. MMR: An integrated massive model rendering system using geometric and image-based acceleration. In *ACM Symposium on Interactive 3D Graphics (I3D)*, pages 199–206. ACM Press, April 1999.

[AD01a] Pierre Alliez and Mathieu Desbrun. Progressive compression for lossless transmission of triangle meshes. In *SIGGRAPH 2001 Conference Proceedings*, pages 198–205, 2001.

[AD01b] Pierre Alliez and Mathieu Desbrun. Valence-driven connectivity encoding of 3d meshes. In *Eurographics 2001 Conference Proceedings*, pages 480–489, September 2001.

[ADF01] C. Andujar and L. De Floriani. Optimal encoding of triangulated polygons. Technical Report DISI-TR-01-06, Department of Computer and Information Science, University of Genova (Italy), 2001.

[AHMS94] E.M. Arkin, M. Held, J.S.B. Mitchell, and S.S. Skiena. Hamiltonian triangulation for fast rendering. In *Second Annual European Symposium on Algorithms*, volume 855, pages 36–47. Springer-Verlag, 1994.

[AS94] P.K. Agarwal and S. Suri. Surface approximation and geometric partitions. In *Proceedings 5th ACM-SIAM Symposium On Discrete Algorithms*, pages 24–33, 1994.

[Ban70] T. Banchoff. Critical points and curvature for embedded polyhedral surfaces. *American Mathematical Monthly*, 77(5):475–485, 1970.

[Bau72] B. G. Baumgart. Winged edge polyhedron representation. Technical Report CS-TR-72-320, Stanford University, Department of Computer Science, October 1972.

[Bau75] B. G. Baumgart. A polyhedron representation for computer vision. In *Proceedings AFIPS National Computer Conference*, volume 44, pages 589–596, 1975.

[BDF90] E. Bruzzone and L. De Floriani. Two data structures for constructing tetrahedralizations. *The Visual Computer: International Journal of Computer Graphics*, 6(5):266–283, 1990.

[BDFM95] M. Bertolotto, L. De Floriani, and P. Marzano. Pyramidal simplicial complexes. In *Proceedings 4th International Symposium on Solid Modeling*, pages 153–162, Salt Lake City, Utah, U.S.A., May 17-19 1995. ACM Press.

[BEHP03] P.-T. Bremer, H. Edelsbrunner, B. Hamann, and V. Pascucci. A multiresolution data structure for two-dimensional Morse functions. In G. Turk, J. van Wijk, and R. Moorhead, editors, *Proceedings IEEE Visualization 2003*, pages 139–146. IEEE Computer Society, October 2003.

[Ben75] J.L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[Bey95] J. Bey. Tetrahedral mesh refinement. *Computing*, 55:355–378, 1995.

[BKSS90] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *Proceedings of the ACM SIGMOD Conference*, pages 322–331, June 1990.

[BL79] S. Beucher and C. Lantuejoul. Use of watersheds in contour detection. In *International Workshop on Image Processing: Real-Time Edge and Motion Detection/Estimation, Rennes, France*, September 17-21, 1979.

[BM98] A. Bieniek and A. Moga. A connected component approach to the watershed segmentation. In H. Heijmans and J. Roerdink, editors, *Mathematical Morphology and its Application to Image and Signal Processing*, pages 215–222. Kluwer Acad. Publ., Dordrecht, 1998.

[Bot98] R. Bott. Morse theory indomitable. *Publ. Math. I. H. E. S.*, 68:99–117, 1998.

[BPS98] C. L. Bajaj, V. Pascucci, and D. R. Shikore. Visualization of scalar topology for structural enhancement. In *Proceedings IEEE Visualization'98*, pages 51–58. IEEE Computer Society, 1998.

[BPTZ99] C. L. Bajaj, V. Pascucci, D. Thompson, and X. Y. Zhang. Parallel accelerated isocontouring for out-of-core visualization. In *Proceedings of the 1999 IEEE Symposium on Parallel visualization and graphics*, pages 97–104. ACM Press, 1999.

[Bri89] E. Brisson. Representing geometric structures in D dimensions: Topology and order. In *Proceedings 5th ACM Symposium on Computational Geometry*, pages 218–227. ACM Press, 1989.

[Bro00] D. Brodsky. Model simplification through refinement. In *In Graphics Interface 2000*, pages 221–228, Ontario, 2000. Canadian Information Processing Society.

[BS96] C. L. Bajaj and D.R. Schikore. Error bounded reduction of triangle meshes with multivariate data. *SPIE*, 2656:34–45, 1996.

[BS98] C. L. Bajaj and D. R. Shikore. Topology preserving data simplification with error bounds. *Computers and Graphics*, 22(1):3–12, 1998.

[BSW83] R. E. Bank, A. H. Sherman, and A. Weiser. Refinement algorithms and data structures for regular local mesh refinement. In R. Stepleman, M. Carver, R. Peskin, W. F. Ames, and R. Vichnevetsky, editors, *Scientific Computing, IMACS Transactions on Scientific Computation*, volume 1, pages 3–17. North-Holland, 1983.

[BYG96] R. Bar-Yehuda and C. Gotsman. Time/space tradeoffs for polygon mesh rendering. *ACM Transactions on Graphics*, 15(2):141–152, April 1996.

[CCM+00] P. Cignoni, D. Costanza, C. Montani, C. Rocchini, and R. Scopigno. Simplification of tetrahedral volume data with accurate error evaluation. In *Proceedings IEEE Visualization 2000*, pages 85–92. IEEE Computer Society, 2000.

[CCMS97] A. Ciampalini, P. Cignoni, C. Montani, and R. Scopigno. Multiresolution decimation based on global error. *The Visual Computer*, 13(5):228–246, 1997.

[CDFM+94] P. Cignoni, L. De Floriani, C. Montani, E. Puppo, and R. Scopigno. Multiresolution modeling and rendering of volume data based on simplicial complexes. In *Proceedings 1994 Symposium on Volume Visualization*, pages 19–26. ACM Press, October 17-18 1994.

[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(1):29–45, January-February 2004.

[CDL+04] P. Cignoni, L. De Floriani, P. Lindstrom, V. Pascucci, J. Rossignac, and C. Silva. Multi-resolution modeling, visualization and streaming of volume meshes. In *Eurographics 2004 - Tutorial*, 2004.

[CFPD04] Lidija Comíc, Leila De Floriani, Laura Papaleo, and Emanuele Danovaro. Morphology-based representation of topographic surfaces: a survey. Technical Report DISI-TR-04-12, Department of Computer Science University of Genova, December 2004.

[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 polygonal models. *ACM Transactions on Graphics*, 23(3), August 2004. Proceedings SIGGRAPH 2004.

[CHHH03] C. Co, B. Heckel, H. Hagen, and B. Hamann. Hierarchical clustering for unstructured volumetric scalar fields. In *Proceedings IEEE Visualization 2003*, pages 325–332, Seattle, October 22-24 2003. IEEE Computer Society.

[Cho97] M.M. Chow. Optimized geometry compression for real-time rendering. In R. Yagel and H. Hagen, editors, *IEEE Visualization '97 Proceedings*, pages 347–354. IEEE Computer Society, 1997.

[CKS98] S. Campagna, L. Kobbelt, and H.-P. Seidel. Directed edges - a scalable representation for triangle meshes. *ACM Journal of Graphics Tools*, 3(4):1–12, 1998.

[Cla76] J. Clark. Hierarchical geometric models for visible surface algorithms. *Communications of the ACM*, 19(10):547–554, October 1976.

[CM02] P. Chopra and J. Meyer. Tetfusion: an algorithm for rapid tetrahedral mesh simplification. In *Proceedings IEEE Visualization 2002*, pages 133–140. IEEE Computer Society, October 2002.

[CMPS97] P. Cignoni, C. Montani, E. Puppo, and R. Scopigno. Multiresolution modeling and visualization of volume data. *IEEE Transactions on Visualization and Computer Graphics*, 3(4):352–369, 1997.

[CMRS03a] 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(4):525–537, November 2003.

[CMRS03b] 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(4):525–537, November 2003.

[CR04] V. Coors and J. Rossignac. Delphi: geometry-based connectivity prediction in triangle mesh compression. *The Visual Computer, International Journal of Computer Graphics*, 20(8-9):507–520, November 2004.

[CS97] Y. Chiang and C. T. Silva. I/O optimal isosurface extraction. In R. Yagel and H. Hagen, editors, *Proceedings IEEE Visualization '97*, pages 293–300. IEEE Computer Society, October 1997.

[CS98] Y. Chiang and C. T. Silva. Interactive out-of-core isosurface extraction. In *Proceedings IEEE Visualization '98*, pages 167–174. IEEE Computer Society, October 1998.

[CS99] Y. Chiang and C. T. Silva. External memory algorithms and visualization. In J. M. Abello and J. S. Vitter, editors, *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, volume 50, pages 247–277. American Mathematical Society, 1999.

[CVM+96] J. Cohen, A. Varshney, D. Manocha, G. Turk, H. Weber, P. Agarwal, F. Brooks, and W. Wright. Simplification envelopes. In *Computer Graphics Proc., Annual Conf. Series (SIGGRAPH '96), ACM Press*, pages 119–128, Aug. 6-8 1996.

[dBSvKO97] Mark de Berg, Otfried Schwarzkopf, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 1997.

[DDF02] E. Danovaro and L. De Floriani. Half-edge multi-tessellation: A compact representations for multi- resolution tetrahedral meshes. In G. Cortellazzo and C. Guerra, editors, *Proceedings First International Symposium on 3D Data Processing Visualization and Transmission*, pages 494–499, June 2002.

[DDFLS02a] E. Danovaro, L. De Floriani, M. Lee, and H. Samet. Multi-resolution tetrahedral meshes: an analysis and a comparison. In G. Wyvill, editor, *Proceedings Shape Modeling International 2002*, pages 83–91, May 2002.

[DDFLS02b] E. Danovaro, L. De Floriani, M. Lee, and H. Samet. Multi-resolution tetrahedral meshes: an analysis and a comparison. In *Proceedings 2002 International Conference on Shape Modeling*, pages 83–91. IEEE Computer Society, May 2002.

[DDFM+03a] E. Danovaro, L. De Floriani, P. Magillo, M. M. Mesmoudi, and E. Puppo. Morphology-driven simplification and multi-resolution modeling of terrains. In E. Hoel and P. Rigaux, editors, *Proceedings ACM-GIS 2003 - The 11th International Symposium on Advances in Geographic Information Systems*, pages 63–70. ACM Press, November 2003.

[DDFM03b] E. Danovaro, L. De Floriani, and M. M. Mesmoudi. Topological analysis and characterization of discrete scalar fields. In T. Asano, R. Klette, and C. Ronse, editors, *Theoretical Foundations of Computer Vision, Geometry, Morphology, and Computational Imaging*, pages 386–402. Springer Verlag, 2003.

[DDFMP01a] E. Danovaro, L. De Floriani, P. Magillo, and E. Puppo. Compressing multiresolution triangle meshes. In C. S. Jensen, M. Schneider, B. Seeger, and V. J. Tsotras, editors, *Advances in Spatial and Temporal Databases*, Lecture Notes in Computer Science, 2121, pages 345–364. Springer Verlag, July 2001.

[DDFMP01b] E. Danovaro, L. De Floriani, P. Magillo, and E. Puppo. Representing vertex-based simplicial multi-complexes. In G. Bertrand, A. Imiya, and R. Klette, editors, *Digital and Image Geometry*, Lecture Notes in Computer Science, N.2243, pages 129–149. Springer Verlag, 2001.

[DDFMP02] E. Danovaro, L. De Floriani, P. Magillo, and E. Puppo. Compact vertex-based multi-resolution simplicial complexes. In G. Bertrand, A. Imiya, and R. Klette, editors, *Digital and Image Geometry*, Lecture Notes in Computer Science. Springer Verlag, 2002.

[DDFMP03] E. Danovaro, L. De Floriani, P. Magillo, and E. Puppo. Data structures for 3d multi-tessellations: An overview. In F. H. Post, G. P. Bonneau, and G. M. Nielson, editors, *Proceedings of the Dagstuhl Scientific Visualization Seminar*, pages 239–256. Kluwer Academic Publishers, 2003.

[Dee95] M. Deering. Geometry compression. In *Computer Graphics Proceedings, Annual Conference Series (SIGGRAPH '95), ACM Press*, pages 13–20, 1995.

[DF89] L. De Floriani. A pyramidal data structure for triangle-based surface description. *IEEE Computer Graphics and Applications*, 9(2):67–78, March 1989.

[DFGH04] L. De Floriani, D. Greenfieldboyce, and A. Hui. A data structure for non-manifold simplicial d-complexes. In *Proceedings ACM/Eurographics Symposium on Geometry Processing*, Nice, France, July 2004.

[DFH04] L. De Floriani and A. Hui. Update operations on 3d simplicial decompositions of non-manifold objects. In *9th ACM Symposium on Solid Modeling and Applications*, pages 169–180. ACM Press, June 2004.

[DFKP04] L. De Floriani, L. Kobbelt, and E. Puppo. A survey on data structures for level-of-detail models. In N. Dodgson, M. Floater, and M. Sabin, editors, *MINGLE Multi-resolution in Geometric Modeling*. Springer Verlag, 2004.

[DFL03] L. De Floriani and M. Lee. Selective refinement on nested tetrahedral meshes. In G. Brunett, B. Hamann, and H. Mueller, editors, *Geometric Modeling for Scientific Visualization*. Springer Verlag, 2003.

[DFM02] L. De Floriani and P. Magillo. Multi-resolution mesh representation: Models and data structures. In M. Floater, A. Iske, and E. Quak, editors, *Principles of Multi-resolution Geometric Modeling*, Lecture Notes in Mathematics, pages 364–418. Springer Verlag, 2002.

[DFM03] L. De Floriani and P. Magillo. Triangle-based multi-resolution models for height fields. In A. Cohen, J. L. Merrien, and L. L. Schumaker, editors, *Curve and Surface Fitting: Saint-Malo 2002*, pages 97–106. Nashboro Press, 2003.

[DFM+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*, June, 15-17 to appear - 2005.

[DFMD02] L. De Floriani, M. M. Mesmoudi, and E. Danovaro. Smale-like decomposition for discrete scalar fields. In R. Kasturi, D. Laurendeau, and C. Suen, editors, *Proceedings International Conference on Pattern Recognition (ICPR)*, volume I, pages 184–187, August 2002.

[DFMMP00] L. De Floriani, P. Magillo, F. Morando, and E. Puppo. Dynamic view-dependent multi-resolution on a client-server architecture. *Computer Aided Design*, 32(13):805–823, 2000.

[DFMMP03] L. De Floriani, M. M. Mesmoudi, F. Morando, and E. Puppo. Non-manifold decompositions in arbitrary dimensions. *CVGIP: Graphical Models*, 65/1-3:2–22, 2003.

[DFMP97a] L. De Floriani, P. Magillo, and E. Puppo. Building and traversing a surface at variable resolution. In *Proceedings IEEE Visualization 97*, pages 103–110. IEEE Computer Society, October 1997.

[DFMP97b] L. De Floriani, P. Magillo, and E. Puppo. Variant - processing and visualizing terrains at variable resolution. In *Proceedings 5th ACM Workshop on Advances in Geographic Information Systems*, November 1997.

[DFMP98] L. De Floriani, P. Magillo, and E. Puppo. Efficient implementation of multi-triangulations. In *Proceedings IEEE Visualization'98*, pages 43–50. IEEE Computer Society, October 1998.

[DFMP99] L. De Floriani, P. Magillo, and E. Puppo. Multi-resolution representation of shapes based on cell complexes. In G. Bertrand, M. Couprie, and L. Perroton, editors, *Discrete Geometry for Computer Imagery, Lecture Notes in Computer Science*, volume 1568, pages 3–18. Springer Verlag, 1999.

[DFMPS02] L. De Floriani, P. Magillo, E. Puppo, and D. Sobrero. A multi-resolution topological representation for non-manifold meshes. In *Proceedings 7th ACM Symposium on Solid Modeling and Applications (SM02)*, June 2002.

[DFP95] L. De Floriani and E. Puppo. Hierarchical triangulation for multi-resolution surface description. *ACM Transactions on Graphics*, 14(4):363–411, October 1995.

[DFPM97] L. De Floriani, E. Puppo, and P. Magillo. A formal approach to multi-resolution modeling. In W. Strasser, R. Klein, and R. Rau, editors, *Geometric Modeling: Theory and Practice*, pages 302–323. Springer-Verlag, 1997.

[DL89] D. Dobkin and M. Laszlo. Primitives for the manipulation of three-dimensional subdivisions. *Algorithmica*, 5(4):3–32, 1989.

[DP02] C. DeCoro and R. Pajarola. XFASTMESH: Fast view-dependent meshing from external memory. In *Proceedings IEEE Visualization 2002*, pages 263–270. IEEE Computer Society, October 2002.

[DS92] H. Desaulnier and N. Stewart. An extension of manifold boundary representation to r-sets. *ACM Transactions on Graphics*, 11(1):40–60, 1992.

[DWS+97] M. Duchaineau, M. Wolinsky, D. E. Sigeti, M. C. Miller, C. Aldrich, and M. B. Mineev-Weinstein. ROAMing terrain: Real-time optimally adapting meshes. In R. Yagel and H. Hagen, editors, *Proceedings IEEE Visualization'97*, pages 81–88. IEEE Computer Society, October 1997.

[Ede87] H. Edelsbrunner. *Algorithms in Combinatorial Geometry*. Springer Verlag, Berlin, 1987.

[EHNP03] H. Edelsbrunner, J. Harer, V. Natarajan, and V. Pascucci. Morse-Smale complexes for piecewise linear 3-manifolds. In *Proceedings 19th ACM Symposium on Computational Geometry*, pages 361–370, 2003.

[EHZ01] H. Edelsbrunner, J. Harer, and A. Zomorodian. Hierarchical Morse complexes for piecewise linear 2-manifolds. In *Proceedings 17th ACM Symposium on Computational Geometry*, pages 70–79. ACM Press, 2001.

[EKT01] W. Evans, D. Kirkpatrick, and G. Townsend. Right-triangulated irregular networks. *Algorithmica*, 30(2):264–286, 2001.

[EMB01] C. Erikson, D. Manocha, and W. Baxter. HLODs: Hierarchical simplifications for faster display of massive geometric environments. In *Proceedings ACM Symposium on Interactive 3D Graphics*, 2001.

[ESAV99] Jihad El-Sana, Elvir Azanli, and Amitabh Varshney. Skip strips: maintaining triangle strips for view-dependent rendering. In *Proceedings of IEEE Visualization '99*, pages 131–138. IEEE Computer Society Press, 1999.

[ESB02] J. El-Sana and E. Bachmat. Optimized view-dependent rendering for large polygonal datasets. In *Proceedings IEEE Visualization 2002*, pages 77–84. IEEE Computer Society, October 2002.

[ESC00] J. El-Sana and Y. Chiang. External memory view-dependent simplification. *Computer Graphics Forum*, 19(3):C139–C150, 2000.

[ESS03] J. El-Sana and N. Sokolovsky. View-dependent rendering for large polygonal models over networks. *International Journal of Image and Graphics*, 3(2):265–290, 2003.

[ESV96] F. Evans, S. Skiena, and A Varshney. Completing sequential triangulations is hard. Technical report, Dept of Computer Science, State University of New York, 1996.

[ESV99] J. El-Sana and A. Varshney. Generalized view-dependent simplification. *Computer Graphics Forum*, 18(3):C83–C94, 1999.

[FAT99] I. Fujishiro, T. Azuma, and Y. Takeshima. Automating transfer function design for comprehensible volume rendering based on 3D field topology analysis. In D. S. Ebert, M. Gross, and B. Hamann, editors, *Proceedings IEEE Visualization '99*, pages 467–470. IEEE Computer Society, 1999.

[FATT00] I. Fujishiro, T. Azuma, Y. Takeshima, and S. Takahashi. Volume data mining using 3D field topology analysis. *IEEE Computer Graphics and Applications*, 20(5):46–51, September-October 2000.

[FMSW00] R. Farias, J. B. Mitchell, C. T. Silva, and B. Wylie. Time critical rendering of irregular grids. In *Proceedings Sibgrapi - XIII Brazilian Symposium on Computer Graphics and Image Processing*, pages 243–250, 2000.

[For98] R. Forman. Morse theory for cell complexes. *Advances in Mathematics*, 134:90–145, 1998.

[FR92] B. Falcidieno and O. Ratto. Two-manifold cell-decomposition of R-sets. In A. Kilgour and L. Kjelldahl, editors, *Proceedings Computer Graphics Forum (EUROGRAPHICS '92)*, volume 11, pages 391–404, September 1992.

[FS01] R. Farias and C. T. Silva. Out-of-core rendering of large, unstructured grids. *IEEE Computer Graphics and Applications*, 21(4):42–50, 2001.

[Gar99] M. Garland. Multi-resolution modeling: Survey and future opportunities. In *Eurographics '99 – State of the Art Reports*, pages 111–131. Eurographics Association, 1999.

[GCP90] E. L. Gursoz, Y. Choi, and F. B. Prinz. Vertex-based representation of non-manifold boundaries. In M. J. Wozny, J. U. Turner, and K. Preiss, editors, *Geometric Modeling for Product Engineering*, pages 107–130. Elsevier Science Publishers B. V., 1990.

[GDL+02] B. Gregorski, M. Duchaineau, P. Lindstrom, V. Pascucci, and K. Joy. Interactive view-dependent rendering of large isosurfaces. In *Proceedings IEEE Visualization 2002*. IEEE Computer Society, October 2002.

[Ger03] T. Gerstner. Multi-resolution visualization and compression of global topographic data. *GeoInformatica*, 7(1):7–32, 2003.

[GG00] G. Greiner and R. Grosso. Hierarchical tetrahedral-octahedral subdivision for volume visualization. *The Visual Computer*, 16:357–369, 2000.

[GGS99] S. Gumhold, S. Guthe, and W. Straßer. Tetrahedral mesh compression with the cut-border machine. In *Proceedings IEEE Visualization'99*, pages 51–58. IEEE Computer Society, 1999.

[GH97] M. Garland and P.S. Heckbert. Surface simplification using quadric error metrics. In *Computer Graphics Proceedings, Annual Conference Series (SIGGRAPH '97), ACM Press*, pages 209–216, 1997.

[GLE97] R. Gross, C. Luerig, and T. Ertl. The multilevel finite element method for adaptive mesh optimization and visualization of volume data. In R. Yagel and H. Hagen, editors, *Proceedings IEEE Visualization '97*, pages 387–394, October 1997.

[GP74] V. Guillemin and A. Pollack. *Differential Topology*. Englewood Cliffs, 1974.

[GP00] T. Gerstner and R. Pajarola. Topology-preserving and controlled topology simplifying multi- resolution isosurface extraction. In *Proceedings IEEE Visualization 2000*, pages 259–266, 2000.

[GR99] T. Gerstner and M. Rumpf. Multiresolutional parallel isosurface extraction based on tetrahedral bisection. In *Proceedings 1999 Symposium on Volume Visualization*. ACM Press, 1999.

[Gri76] H. B. Griffiths. *Surfaces*. Cambridge University Press, 1976.

[GS85] L. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and computation of Voronoi diagrams. *ACM Transactions on Graphics*, 4(2):74–123, April 1985.

[GS98a] M. H. Gross and O. G. Staadt. Progressive tetrahedralizations. In *Proceedings IEEE Visualization'98*, pages 397–402. IEEE Computer Society, 1998.

[GS98b] S. Gumhold and W. Straßer. Real time compression of triangle mesh connectivity. In *ACM Computer Graphics Proceedings, (SIGGRAPH '98)*, pages 133–140, July 1998.

[GS02] M. Garland and E. Shaffer. A multiphase approach to efficient surface simplification. In *Proceedings of the conference on Visualization '02*, pages 117–124. IEEE Computer Society, October 2002.

[GTLH98] A. Gueziec, G. Taubin, F. Lazarus, and W. Horn. Converting sets of polygons to manifold surfaces by cutting and stitching. In *Conference abstracts and applications: SIGGRAPH 98*, Computer Graphics, pages 245–245. ACM Press, 1998.

[Gué95] A. Guéziec. Surface simplification with variable tolerance. In *Second Annual International Symposium on Medical Robotics and Computer Assisted Surgery*, pages 132–139, Baltimore, MD, November 1995.

[Gum00] Stefan Gumhold. New bounds on the encoding of planar triangulations. Technical Report WSI–2000–1, Wilhelm-Schickard-Institut für Informatik, University of Tübingen, Germany, January 2000.

[HC94] B. Hamann and J. L. Chen. Data point selection for piecewise trilinear approximation. *Computer Aided Geometric Design*, 11:477–489, 1994.

[Heb94] D. J. Hebert. Symbolic local refinement of tetrahedral grids. *Journal of Symbolic Computation*, 17(5):457–472, May 1994.

[Hop96] H. Hoppe. Progressive meshes. In *ACM Computer Graphics Proceedings, Annual Conference Series, (SIGGRAPH'96)*, pages 99–108, 1996.

[Hop97] H. Hoppe. View-dependent refinement of progressive meshes. In *ACM Computer Graphics Proceedings, Annual Conference Series, (SIGGRAPH'97)*, pages 189–198, August 1997.

[Hop98a] H. Hoppe. Efficient implementation of progressive meshes. *Computer & Graphics*, 22(1):27–36, 1998.

[Hop98b] H. Hoppe. Smooth view-dependent level-of-detail control and its application to terrain rendering. In *Proceedings IEEE Visualization'98*, pages 35–42. IEEE Computer Society, 1998.

[Hop99] Hugues Hoppe. New quadric metric for simplifying meshes with appearance attributes. In *Proceedings of IEEE Visualization '99*, pages 59–66. IEEE Computer Society Press, 1999.

[IG03] M. Isenburg and S. Gumhold. Out-of-core compression for gigantic polygon meshes. *ACM Transactions on Graphics*, 22(3):935–942, July 2003.

[ILGS03] M. Isenburg, P. Lindstrom, S. Gumhold, and J. Snoeyink. Large mesh simplification using processing sequences. In *Proceedings of the Visualization 2003 Conference*, pages 465–472. IEEE Computer Society Press, October 2003.

[IS01] M. Isenburg and J. Snoeyink. Spirale Reversi: Reverse decoding of the Edgebreaker encoding. *Computational Geometry*, 20(1):39–52, October 2001.

[KCVS98] L. Kobbelt, S. Campagna, J. Vorsatz, and H.P. Seidel. Interactive multiresolution modeling of arbitrary meshes. In *Computer Graphics Proceedings, Annual Conference Series (SIGGRAPH '98), ACM Press*, 1998.

[Kel55] John L. Kelley. *General Topology*. Springer-Verlag, New York, NY, USA, 1955.

[KG98] R. Klein and S. Gumhold. Data compression of multi-resolution surfaces. In *Visualization in Scientific Computing '98*, pages 13–24. Springer Verlag, 1998.

[KL01] J. Kim and S. Lee. Truly selective refinement of progressive meshes. In *Proc. Graphics Interface 2001*, pages 101–110, 2001.

[KLK04] J. Kim, S. Lee, and L. Kobbelt. View-dependent streaming of progressive meshes. In *Proc. of International Conference on Shape Modeling and Applications (SMI 2004)*, pages 209–220. IEEE Computer Society, 7-9 June 2004.

[Kob00] L. Kobbelt. $\sqrt{3}$ subdivision. In *Proceedings of ACM SIGGRAPH 2000*, pages 103–112. ACM, 2000.

[KR99] D. King and J. Rossignac. Guaranted 3.67v bit encoding of planar triangle graphs. In *11th Canadian Conference on Computational Geometry*, Vancouver, August 15-18 1999.

[Law77] C. L. Lawson. Software for C1 Surface Interpolation. In J. R. Rice, editor, *Mathematical Software III*, pages 161–164. Academic Press, 1977.

[LDFS01] M. Lee, L. De Floriani, and H. Samet. Constant time neighbor finding in hierarchical tetrahedral meshes. In *Proceedings International Conference on Shape Modeling*, pages 286–295, May 2001.

[LE97] D. Luebke and C. Erikson. View-dependent simplification of arbitrary polygonal environments. In *ACM Computer Graphics Proceedings, Annual Conference Series, (SIGGRAPH '97)*, pages 199–207, 1997.

[LGCR04] J. Lluch, R. Gaitan, E. Camahort, and R.Vivó. Interactive three-dimensional rendering on mobile computer devices. In *II Ibero-American Symp. on Computer Graphics, SIACG*, 17-20 October 2004.

[Lie91] P. Lienhardt. Topological models for boundary representation: a comparison with n-dimensional generalized maps. *Computer Aided Design*, 23(1):59–82, 1991.

[Lin00] P. Lindstrom. Out-of-core simplification of large polygonal models. In *ACM Computer Graphics (SIGGRAPH 2000 Proceedings)*, pages 259–270. ACM Press, July 2000.

[Lin03] P. Lindstrom. Out-of-core construction and visualization of multi-resolution surfaces. In *ACM SIGGRAPH 2003 Symposium on Interactive 3D Graphics*, pages 93–102. ACM Press, April 2003.

[LKR+96] P. Lindstrom, D. Koller, W. Ribarsky, L. F. Hodges, N. Faust, and G. A. Turner. Real-time continuous level of detail rendering of height fields. In *Proceedings SIGGRAPH'96*, pages 109–118, August 1996.

[LL01] S. H. Lee and K. Lee. Partial-entity structure: a fast and compact non-manifold boundary representation based on partial topological entities. In *Proceedings Sixth ACM Symposium on Solid Modeling and Applications*, pages 159–170. Ann Arbor, Michigan, June 2001.

[LP01] P. Lindstrom and V. Pascucci. Visualization of large terrains made easy. In *Proceedings IEEE Visualization 2001*, pages 363–370, October 2001.

[LP02] P. Lindstrom and V. Pascucci. Terrain simplification simplified: A general framework for view- dependent out-of-core visualization. *IEEE Transactions on Visualization and Computer Graphics*, 8(3):239–254, 2002.

[LPC+00] M. Levoy, K. Pulli, B. Curless, S. Rusinkiewics, D. Koller, L. Pereira, M. Gintzon, S. Anderson, J. Davis, J. Gensberg, J. Shade, and D. Fulk. The digital michelangelo project: 3d scanning of large statues. In *ACM Computer Graphics Proceedings, Annual Conference Series, (SIGGRAPH'00)*, pages 131–144, 2000.

[LRC+02] D. Luebke, M. Reddy, J. Cohen, A. Varshney, B. Watson, and R. Huebner. *Level of Detail for 3D Graphics*. Morgan-Kaufmann, 2002.

[LS00] M. Lee and H. Samet. Navigating through triangle meshes implemented as linear quadtrees. *ACM Transactions on Graphics*, 19(2):79–121, April 2000.

[LS01] P. Lindstrom and C. T. Silva. A memory insensitive technique for large model simplification. In *IEEE Visualization 2001*, pages 121–126, 550, October 2001.

[LT97] H. Lopes and G. Tavares. Structural operators for modeling 3-manifolds. In *Proceedings Fourth ACM Symposium on Solid Modeling and Applications*, pages 10–18. ACM Press, May 1997.

[LT99] P. Lindstrom and G. Turk. Evaluation of memoryless simplification. *IEEE Transactions on Visualization and Computer Graphics*, 5(2):98–115, April-June 1999.

[Mag99] P. Magillo. *Spatial Operations on Multiresolution Cell Complexes*. PhD thesis, Dept. of Computer and Information Sciences, University of Genova (Italy), 1999.

[Mag00] P. Magillo. *The MT (Multi-Tessellation) Package*. Dept. of Computer and Information Sciences (DISI), University of Genova, Genova, Italy, January 2000.

[Man87] M. Mantyla. *An Introduction to Solid Modeling*. Computer Science Press, 1987.

[Mas91] William S. Massey. *A basic course in algebraic topology*. Springer-Verlag, New York, NY, USA, 1991.

[Mau95] J. M. Maubach. Local bisection refinement for n-simplicial grids generated by reflection. *SIAM Journal on Scientific Computing*, 16(1):210–227, January 1995.

[MB00] P. Magillo and V. Bertocci. Managing large terrain data sets with a multiresolution structure. In A. M. Tjoa, R. R. Wagner, and Ala Al-Zobaidie, editors, *Proc. 11th Int. Conf. on Database and Expert Systems Applications*, pages 894–898. IEEE, September 2000.

[Mey94] F. Meyer. Topographic distance and watershed lines. *Signal Processing*, 38:113–125, 1994.

[Mil63] J. Milnor. *Morse Theory*. Princeton University Press, 1963.

[MP78] D. E. Mueller and F. P. Preparata. Finding the intersection of two convex polyhedra. *SIAM Theoretical Computer Science*, 7:217–236, 1978.

[MR96] A. Meijster and J. Roerdink. Computation of watersheds based on parallel graph algorithms. In P. Maragos, R. W. Shafer, and M. A. Butt, editors, *Mathematical Morphology and its Application to Image Segmentation*, pages 305–312. Kluwer, 1996.

[MW99] A. Mangan and R. Whitaker. Partitioning 3D surface meshes using watershed segmentation. *Transaction on Visualization and Computer Graphics*, 5(4):308–321, 1999.

[NGH04] X. Ni, M. Garland, and J.C. Hart. Fair morse functions for extracting the topo-logical structure of a surface mesh. *ACM Transaction on Graphics*, 23(3):613–622, 2004.

[Nie97] G. M. Nielson. Tools for triangulations and tetrahedralizations and constructing functions defined over them. In G. M. Nielson, H. Hagen, and H. Müller, editors, *Scientific Visualization: Overviews, Methodologies and Techniques*, chapter 20, pages 429–525. IEEE Computer Society, 1997.

[OR97] M. Ohlberger and M. Rumpf. Hierarchical and adaptive visualization on nested grids. *Computing*, 56(4):365–385, 1997.

[Ore82] J. A. Orenstein. Multidimensional tries used for associative searching. *Information Processing Letters*, 14(4):150–157, 1982.

[Paj98] R. Pajarola. Large scale terrain visualization using the restricted quadtree triangulation. In D. Ebert, H. Hagen, and H. Rushmeier, editors, *Proceedings IEEE Visualization'98*, pages 19–26. IEEE Computer Society, October 1998.

[Paj01] R. Pajarola. FASTMESH: efficient view-dependent meshing. In *Proceedings Pacific Graphics 2001*, pages 22–30. IEEE Computer Society, 2001.

[Pas01] V. Pascucci. On the topology of the level sets of a scalar field. In *Proceedings 12th Canadian Conference on Computational Geometry*, pages 141–144, August 2001.

[Pas02] V. Pascucci. Slow Growing Subdivisions (SGSs) in any dimension: towards removing the curse of dimensionality. *Computer Graphics Forum*, 21(3), 2002.

[Pas03] V. Pascucci. Multi-resolution indexing for hierarchical out-of-core traversal of rectilinear grids. In G. Farin, H. Hagen, and B. Hamann, editors, *Hierarchical and Geometrical Methods for Scientific Visualization*, 2003.

[Pas04] V. Pascucci. Topology diagrams of scalar fields in scientific visualization. In S. Rana, editor, *Topological Data Structures for Surfaces*, pages 121–129. John Wiley & Sons Ltd, 2004.

[PBCF93] A. Paoluzzi, F. Bernardini, C. Cattani, and V. Ferrucci. Dimension-independent modeling with simplicial complexes. *ACM Transactions on Graphics*, 12(1):56–102, January 1993.

[PD75] T. K. Peucker and D. H. Douglas. Detection of Surface-Specific Points by Local Parallel Processing of Discrete Terrain Elevation Data. *Computer Graphics and Image Processing*, 4:375–387, 1975.

[PH97] J. Popovic and H. Hoppe. Progressive simplicial complexes. In *ACM Computer Graphics Proceedings, Annual Conference Series, (SIGGRAPH '97)*, pages 217–224, 1997.

[PR00] R. Pajarola and J. Rossignac. Compressed progressive meshes. *IEEE Transactions on Visualization and Computer Graphics*, 6(1):79–93, 2000.

[Pri00] C. Prince. Progressive meshes for large models of arbitrary topology. Master Thesis, Department of Computer Science and Engineering, University of Washington, August 2000.

[PRS99] R. Pajarola, J. Rossignac, and A. Szymczak. Implant Sprays: Compression of progressive tetrahedral mesh connectivity. In *Proceedings IEEE Visualization'99*, pages 299–305. IEEE Computer Society, 1999.

[PTVF92] W. Press, S. Teukolsky, W. Vetterling, and B. Flannery. *Numerical recipes in c - second edition*. Cambridge University Press, 1992.

[Pup98] E. Puppo. Variable resolution triangulations. *Computational Geometry Theory and Applications*, 11(3-4):219–238, December 1998.

[RB93] J. Rossignac and P. Borrel. Multi-resolution 3D approximations for rendering complex scenes. In T. L. Kunii B. Falcidieno, editor, *Modeling in Computer Graphics*, pages 455–465. Springer-Verlag, October 17-18 1993.

[RC99] J. Rossignac and D. Cardoze. Matchmaker: Manifold BReps for non-manifold R-sets. In W. F. Bronsvoort and D. C. Anderson, editors, *Proceedings Fifth Symposium on Solid Modeling and Applications*, pages 31–41. ACM Press, June 1999.

[RL92] M. Rivara and C. Levin. A 3D refinement algorithm for adaptive and multigrid techniques. *Communications in Applied Numerical Methods*, 8:281–290, 1992.

[RM00] J. Roerdink and A. Meijster. The watershed transform: definitions, algorithms, and parallelization strategies. *Fundamenta Informaticae*, 41:187–228, 2000.

[RO90] J. R. Rossignac and M. A. O'Connor. SGC: A dimension-independent model for point-sets with internal structures and incomplete boundaries. In M. J. Wozny, J. U. Turner, and K. Preiss, editors, *Geometric Modeling for Product Engineering*, pages 145–180. Elsevier Science Publishers B. V. (North–Holland), 1990.

[RO96] K. J. Renze and J. H. Oliver. Generalized unstructured decimation. *IEEE Computational Geometry & Applications*, 16(6):24–32, 1996.

[Ros99] J. Rossignac. Edgebreaker: Connectivity compression for triangle meshes. *IEEE Transactions on Visualization and Computer Graphics*, 5(1), 1999.

[RR96] R. Ronfard and J. Rossignac. Full-range approximation of triangulated polyhedra. *Computer Graphics Forum (Eurographics'96 Proceedings)*, 15(3):67–76, 1996.

[RS99] J. Rossignac and A. Szymczak. Wrap & zip: Linear decoding of planar triangle graphs. *Computational Geometry: Theory and Applications*, 14:119–135, 1999.

[RSS01] J. Rossignac, A. Safonova, and A. Szymczak. 3D compression made simple: Edge-breaker on a corner table. In *Proceedings Shape Modeling International 2001*, May 2001.

[Sal00] D. Salomon. *Data Compression: The Complete Reference*. Springer-Verlag, 2000.

[Sam90] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1990.

[Sam05] H. Samet. *Foundations of Multi-Dimensional Data Structures*. Morgan-Kaufmann, to appear, 2005.

[Sch05] Bernhard Schneider. Extraction of hierarchical surface networks from bilinear surface patches. *Geographical Analysis*, 37:244–263, 2005.

[SDDFM03] N. Sokolovsky, E. Danovaro, L. De Floriani, and P. Magillo. Encoding level-of-detail tetrahedral meshes based on half-edge collapse. In *MINGLE'2003 Proceedings*, pages 91–102. Springer Verlag, September 2003.

[SG01] E. Shaffer and M. Garland. Efficient adaptive simplificationof massive meshes. In *Proceedings IEEE Visualization '01*. IEEE Computer Society, October 2001.

[SG05] E. Shaffer and M. Garland. A multi-resolution representation for massive meshes. *IEEE Transactions on Visualization and Computer Graphics*, to appear, 2005.

[She98] J. Shewchuk. Tetrahedral mesh generation by Delaunay refinement. In *Proceedings of the Fourteenth Annual Symposium on Computational Geometry*, pages 86–95. Association for Computing Machinery, June 1998.

[Sma60] S. Smale. Morse inequalities for a dynamical system. *Bulletin of American Mathematical Society*, 66:43–49, 1960.

[Soi04] P. Soille. *Morphological Image Analysis: Principles and Applications*. Springer-Verlag, Berlin and New York, 2004.

[SP92] L. Scarlatos and T. Pavlidis. Hierarchical triangulation using cartographics coherence. *CVGIP: Graphical Models and Image Processing*, 54(2):147–161, March 1992.

[SR99] Andrzej Szymczak and Jarek Rossignac. Grow & fold: compression of tetrahedral meshes. In *SMA '99: Proceedings of the fifth ACM symposium on Solid modeling and applications*, pages 54–64, New York, NY, USA, 1999. ACM Press.

[SS00] S. L. Stoev and W. Strasser. Extracting regions of interest applying a local watershed transformation. In *Proceedings IEEE Visualization'00*, pages 21–28. ACM Press, 2000.

[SW04] B. Schneider and J. Wood. Construction of metric surface networks from raster-based DEMs. In S. Rana, editor, *Topological Data Structures for Surfaces*, pages 53–70. John Wiley & Sons Ltd, 2004.

[SZL92] W. J. Schroeder, J. A. Zarge, and W. E. Lorensen. Decimation of triangle meshes. *Computer Graphics (SIGGRAPH '92 Proc.)*, 26(2):65–70, July 1992.

[TF78] J. Toriwaki and T. Fukumura. Extraction of Structural Information from Gray Pictures. *Computer Graphics and Image Processing*, 7:30–51, 1978.

[TG98] C. Touma and C. Gotsman. Triangle mesh compression. In *Proceedings Graphics Interface'98*, pages 26–34, 1998.

[TGHL98] G. Taubin, A. Guéziec, W. Horn, and F. Lazarus. Progressive forest split compression. In *Computer Graphics (SIGGRAPH '98 Proceedings)*, pages 123–132. ACM Press, 1998.

[THJ99] I. J. Trotts, B. Hamann, and K. I. Joy. Simplification of tetrahedral meshes with error bounds. *IEEE Transactions on Visualization and Computer Graphics*, 5(3):224–237, 1999.

[THJW98] I.J. Trotts, B. Hamann, K.I. Joy, and D.F. Wiley. Simplification of tetrahedral meshes. In *Proceedings IEEE Visualization'98*, pages 287–295, Research Triangle Park, NC, 1998.

[TIKU95] S. Takahashi, T. Ikeda, T. L. Kunii, and M. Ueda. Algorithms for extracting correct critical points and constructing topological graphs from discrete geographic elevation data. *Computer Graphics Forum*, 14(3):181–192, 1995.

[TLHR98] G. Taubin, F. Lazarus, W. Horn, and J. Rossignac. Geometry coding and vrml. *Proceedings of the IEEE*, 96(6):1228–1243, 1998.

[TR98] G. Taubin and J. Rossignac. Geometric compression through topological surgery. *ACM Transactions on Graphics*, 17(2):84–115, 1998.

[Tur84] G. Turán. On the succinct representation of graphs. In *Discrete Applied Mathematics*, volume 8, pages 289–294. North-Holland, 1984.

[Tut62] William T. Tutte. A census of planar triangulations. *Canad. J. Math.*, 14:21–38, 1962.

[VdFG99] L. Velho, L. Henriquez de Figueredo, and J. Gomes. A unified approach for hierarchical adaptive tessellation of surfaces. *ACM Transactions on Graphics*, 4(18):329–360, 1999.

[VG00] L. Velho and J. Gomes. Variable resolution 4-k meshes: Concepts and applications. *Computer Graphics Forum*, 19(4):195–214, 2000.

[VM02] G. Varadhan and D. Manocha. Ouf-of-core rendering of massive geometric environments. In R. Moorhead, M. Gross, and K. I. Joy, editors, *Proceedings IEEE Visualization 2002*, pages 69–76. IEEE Computer Society, October 2002.

[VS91] L. Vincent and P. Soile. Watershed in digital spaces: An efficient algorithm based on immersion simulation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13(6):583–598, 1991.

[VS94] J.S. Vitter and E.A.M. Shriver. Algorithms for parallel memory i: Two-level memories. *Algorithmica*, 12(2-3):110–147, 1994.

[Wei88] K. Weiler. The radial edge data structure: a topological representation for non- manifold geometric boundary modeling. In H. W. McLaughlin J. L. Encarnacao, M. J. Wozny, editor, *Geometric Modeling for CAD Applications*, pages 3–36. Elsevier Science Publishers B. V. (North–Holland), 1988.

[Wil93] P.L. Williams. *Interactive Direct Volume Rendering of Curvilinear and Unstructured Data*. PhD thesis, University of Illinois at Urbana–Champaign, 1993.

[WLH85] L. T. Watson, T. J. Laffey, and R. M. Haralick. Topographic classification of digital image intensity surfaces using generalized splines and the discrete cosine transformation. *Computer Vision, Graphics, and Image Processing*, 29:143–167, 1985.

[WSHH02] G. H. Weber, G. Schueuermann, H. Hagen, and B. Hamann. Exploring scalar fields using critical isovalues. In *Proceedings IEEE Visualization 2002*, pages 171–178. IEEE Computer Society, 2002.

[WYM98] W. Wang, J. Yang, and R. Muntz. PK-tree: a spatial index structure for high dimensional point data. In K. Tanaka and S. Ghandeharizadeh, editors, *Proceedings of the 5th International Conference on Foundations of Data Organization and Algorithms (FODO)*, pages 27–36, November 1998.

[XESV97] J. C. Xia, J. El-Sana, and A. Varshney. Adaptive real-time level-of-detail-based rendering for polygonal models. *IEEE Transactions on Visualization and Computer Graphics*, 3(2):171–183, 1997.

[YK95] Y. Yamaguchi and F. Kimura. Non-manifold topology based on coupling entities. *IEEE Computer Graphics and Applications*, 15(1):42–50, January 1995.

[ZBB01] X. Zhang, C. Bajaj, and W. Blanke. Scalable isosurface visualization of massive datasets on cots clusters. In *Proceedings of the IEEE 2001 Symposium on parallel and large-data visualization and graphics*, pages 51–58. IEEE Press, 2001.

[ZCK97] Y. Zhou, B. Chen, and A. Kaufman. Multiresolution tetrahedral framework for visualizing regular volume data. In R. Yagel and H. Hagen, editors, *Proceedings IEEE Visualization '97*, pages 135–142, Phoenix, AZ, October 1997.

[Zha95] S. Zhang. Successive subdivision of tetrahedra and multigrid methods on tetrahedral meshes. *Houston J. Mathematics*, 21:541–556, 1995.