Dipartimento di Informatica e
Scienze dell'Informazione

Efficient Representations for Multi-Resolution Modeling

by

Davide Sobrero

Theses Series **DISI-TH-2006-06**

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

Efficient Representations for Multi-Resolution Modeling

by

Davide Sobrero

April, 2006

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 (S.S.D. INF/01)

Submitted by Davide Sobrero
DISI, Università di Genova
sobrero@disi.unige.it

Date of submission: April 2006

Title: Efficient Representations
For Multi-Resolution Modeling

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

Enrico Puppo
DISI, Università di Genova
puppo@disi.unige.it

Ext. Reviewers: Riccardo Scateni
Dipartimento di Matematica e Informatica, Università di Cagliari
riccardo@unica.it

Jihad El-Sana
Department of Computer Science, Ben Gurion University of The Negev
elsana@cs.bgu.ac.il

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. For this reason, Level-of-Detail (LOD) techniques have been proposed for triangle-based models of terrains and of the boundary of 3D manifold objects in a variety of applications. The basic operation that an LOD model needs to support is selective refinement. Selective refinement consists of extracting an adaptive mesh-based representation that satisfies some application-dependent requirements. The extracted representation may have a resolution which is uniform or varies in different parts of the shape or of the field domain.

In this thesis we address the problem of designing LOD data structures for supporting selective refinement. The emphasis of this work is designing and developing compact LOD data structures tailored to specific applications. Our reference LOD model is the Multi-Tessellation, a dimension-independent model based on a collection of modifications organized according to a dependency relation, which guides the extraction of meshes at different resolution during selective refinement. In our work, we have developed LOD data structures for models generated by applying specific modifications, namely edge collapse or vertex-pair collapse. In both cases, the dependency relation can be effectively described through a forest of binary tree plus a vertex enumeration mechanism resulting in a data structure called a view-dependent tree.

In the thesis, we first address the problem of representing and processing complex 3D objects objects, described by simplicial complexes, which consist of parts of mixed dimensions, and with a non-manifold topology, at different levels of detail. We have designed and developed a new data structure for two-dimensional simplicial complexes embedded in the 3D Euclidean space, that we call the Triangle-Segment data structure, as well as algorithms for navigating and updating simplicial complexes described by such a data structure. We have also developed an LOD Model, that we call a Non-manifold Multi-Tessellation (NMT), that unlike the Multi-Tessellation, can deal also with non-manifold and non-regular objects, and a compact representation for such a model, when generated from 2D simplicial complex through vertex-pair collapse. In this latter, the dependency relation is described through a view-dependent tree.

The second application field we have considered in our work is in the context of volume data visualization, where models are usually very large and operations necessary to support visualization can be computationally intensive. We have developed a data structure

for compact encoding of an LOD model of a three-dimensional scalar field based on unstructured tetrahedral meshes. Such data structure, called an Half-Edge Tree (HET), is built through the iterative application of a half-edge collapse, i.e. by contracting an edge to one of its endpoints. Here, the dependency relation is encoded using a version of the view-dependent tree customized for three dimensional half-edge collapse.

Finally, the last task we have considered in this work is the design of a general framework for the transmission of compact LOD data structures in a network environment, to permit remote users to query and possibly download an LOD model stored on a server. This application field is very interesting for the increasing availability of wide-band transmission technologies, that make possible the transmission of larger amount of data than in the past.

To my families

Table of Contents

Chapter 1

Introduction

The increasing power of modern acquisition devices and computer simulations have produced very large data sets describing spatial objects, scalar and vector fields for different applications domains, such as industrial design, medical imaging, entertainment, and Geographical Information Systems (GISs). Many are the examples, such as 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 hundreds of millions triangles (see, for instance, models from the Digital Michelangelo Project [LPC+00]). 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 large size of available data sets has led, through the years, to an intense research activity on mesh simplification and on *multi-resolution* (aka *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. Unfortunately, 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 can describe an object, or a scalar field, as a mesh of simple basic elements (triangles, tetrahedra, cuboids) at different resolutions, according to the some criteria specified. 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).

Dealing with LOD modeling, an interesting issue is related to the design and implementation of efficient and compact data structures for multi-resolution models for large meshes. In the case of huge dimension meshes that do not fit in primary memory, have been proposed *out-of-core* techniques, to simplify and build multi-resolution models directly on secondary memory. However, day by day, the computing power and the availability of primary memory in commercial systems grow, making possible the manipulation and the visualization of even more large data sets. To this aim, becomes relevant the design and the implementation of compact and efficient encoding of multi-resolution models.

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. Moreover, especially in CAD applications, it is possible to have non-manifold models, situation which requires a special attention and leads to a specialization of the MT.

The aim of this thesis is the study and the implementation of compact data structures for two-dimensional meshes embedded in three-dimensional space representing non-manifold shapes, and tetrahedral meshes for visualization for three-dimensional shapes and volume data analysis and visualization. In particular, we have considered compact encoding of the MT in which the partial ordering of the modification is encoded with the *view-dependent tree* proposed by El-Sana and Varshney in [ESV99] or extended versions.

To this aim, we have identified three tasks:

- design and development of a multi-resolution model for non-regular non-manifold two-dimensional simplicial meshes describing complex shapes;
- design and development of a compact data structure for tetrahedral multi-resolution models describing three-dimensional scalar fields;
- design of a client/server framework for transmission and manipulation of compact multi-resolution in network environment.

The first task was completed with the proposal of the *Non-Manifold MT* (NMT), a multi-resolution model for non-regular non-manifold 2D meshes, based on a compact encoding of

the modifications and characterized by the use of the view-dependent tree for encoding the partial order among the modifications. Inside this work, we have also defined a new data structure non-regular non-manifold mesh, called *Triangle Segment* (TS) data structure. The TS have been implemented and has been made available in the tool repository of the European Network of Excellence AIM@SHAPE [oEA].

We have faced the second task by focusing on compact representations for multi-resolution models for three-dimensional scalar fields. Starting from [CDFM+04], we have studied a compact multi-resolution model for tetrahedral meshes built through the *half-edge collapse* operation, adapting the original view-dependent tree, which was proposed for *full-edge collapse*, in a new version called the *Half Edge Tree* (HET).

For the third task, we start analyzing the problem of transmitting and querying from remote a multi-resolution model resident on a server. The most popular and interesting applications in network environment are related to the World Wide Web context: working in this direction, we was interested in designing a system in which one server con easily manage several connections with different clients. We have studied a new approach assuming that the client is not a simple visualization machine, but it can store part of the informations related to the multi-resolution model and perform itself the query. We propose a general framework for network transmission of a compact multi-resolution data structure, with the dependency relation encoded as view-dependent tree or its variants.

This thesis is organized into seven chapters. A brief description of the content of each chapter is reported below.

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 the *Multi-Tesselletion* model, which is the basis of the work of this thesis.

Chapter 3 presents the state of the art in the specific fields related to the subject of this thesis. The first part is devoted to review state-of-the-art data structures for discrete and variable-resolution LOD models, both for regular and irregular datasets. In the second part, we review data structures for non-manifold two-dimensional meshes. Finally, we briefly review works in the field of transmission over the networks of multi-resolution models, from progressive models to more recent client/server approaches.

In Chapter 4, we present the data structure we have designed and implemented to represent at different resolutions two-dimensional non regular non-manifold objects, which we have proposed in [DFMPS02] and, in a extended version in [DFMPS04]. In this Chapter, we also describe the new data structure we have designed for non-regular non-manifold 2D meshes.

In Chapter 5, we describe and analyze a compact data structure we have developed for mesh-based multi-resolution models generated from unstructured tetrahedral meshes

through specific refinement or coarsening operator. Selecting a specific operator results in a loss of generality with respect to the Multi-Tessellation, 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 multi-resolution tetrahedral meshes built through half-edge collapse [DFM+05b], 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 based on the full-edge collapse operation, described in [CDFM+04].

Chapter 6 presents the general framework we have designed for the transmission over the network of a compact multi-resolution model. Since one of the most common network environment is the World Wide Web, we have defined a general framework that can be easily generalized from the client-server approach with one client, described in this chapter, to a multi-clients system, as it is the case in WEB applications.

Chapter 2

Background Notions

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. In Section 2.2 we present the *Multi-Tessellation* model, which is at the fundamental basis of this work.

2.1 Cell and Simplicial Complexes

In this Section, we introduce the definitions of cell and simplicial complexes, their properties and the topological relations which are the basis for defining data structures which encode them. 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 [Man88]. 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 any 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 the *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.

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 that of $\mathbb{E}^n$.

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 which is 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 $\mathbb{R}^n$.

2.1.3 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 The Multi-Tesselation

In this Section, we review the *Multi-Tessellation*, a variable-resolution continuous Level-Of-Detail (LOD) model which can represent multi-resolution simplicial meshes in arbitrary dimensions whit a manifold domain [DFPM97]. The Multi-Tessellation can also be seen as a general framework, that allows represent variable-resolution LOD models for simplicial meshes. In Subsection 2.2.1, we give a formal definition of such framework, and we report desirable properties (see [DFPM97, Mag99, Dan05] for more details).

2.2.1 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, the notion of validity of a modification for a simplicial complex can be defined as follows:

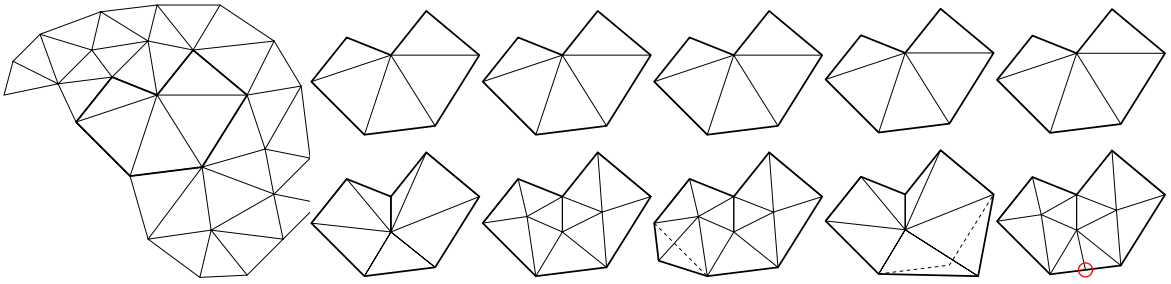


Figure 2.1: Five modifications on a simplicial complex: (a), (b) and (c) are valid while (d) and (e) are not valid since (d) violates condition 2 and (e) violates condition 3 in Definition 2.2.1

Definition 2.2.1 *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 the boundary of the hole created in Σ when removing Σ_{old}.*

Examples of valid and invalid modifications are shown in Figure 2.1. It has been proven 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 multi-resolution 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 $\mathcal{S} = [u_0, \dots u_m]$ 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, a modification is defined 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 2.2.2 *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.

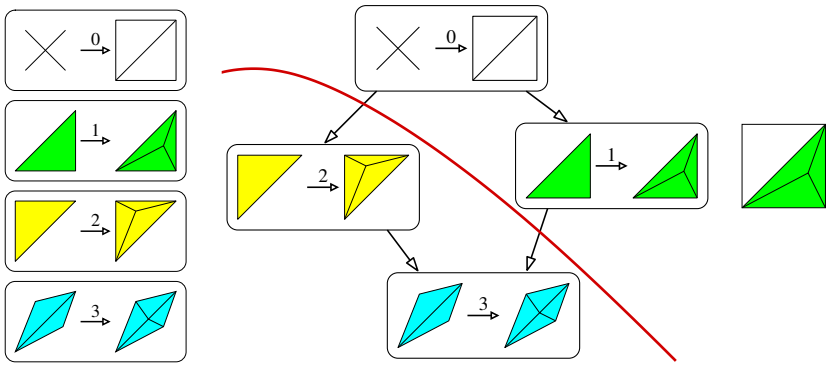


Figure 2.2: A sequence of modifications (a), an MT with direct dependencies represented with arrows (b), and the mesh corresponding to the front represented by the bold red line (c).

- *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 *Directed Acyclic Graph* (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 2.2 shows a sequence of modifications, the corresponding Multi-Tessellation, and the front simplicial complex associated to the front in Figure 2.2.

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 modifications 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.

2.2.2 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 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 multi-resolution 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.

2.2.3 Selective Refinement Queries

Selective refinement queries consist of extracting from an LOD model a mesh satisfying some application-dependent requirements based on the 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.

2.2.3.1 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).

2.2.3.2 Standard LOD 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 multi-resolution model whose modifications have an error value associated with. The first one selects a subset of modifications that produce a mesh at uniform resolution 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 iso-contour) 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 The 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 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 mesh at full resolution in the focus region. 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.

Chapter 3

State of the Art

This Chapter presents the state of the art in several fields related to the subject of this thesis, namely multi-resolution representations for two- and three-dimensional scalar fields described by simplicial meshes, data structures for boundary representation of non-manifold objects and multi-resolution data structures in network environment.

The Chapter is organized in three sections. Section 3.1 reviews state-of-the-art data structures for discrete and continuous multi resolution models, both for regular and irregular datasets. Section 3.2 reviews data structures proposed for two-dimensional cell complexes in the context of modeling non-manifold objects. Finally, Section 3.3 describes state-of-the-art techniques to transmit and query multi-resolution models over a network.

3.1 Multi-resolution Models

A mesh-based *multi-resolution* (aka *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.

In this Section, we review data structures for describing such models. Subsection 3.1.1 describes discrete and continuous multi-resolution models. Specifically, Subsection 3.1.1.1 describes data structures for LOD models based on regular meshes, while Subsection 3.1.1.2 presents data structures for multi resolution models based on irregular meshes. Multi-resolution models for tetrahedral datasets are reviewed in Subsection 3.1.2. An extensive treatment of multi-resolution models for view-dependent visualization and of their applications can be found in [LRC+02, CDL+04], and in [DDFMP03]. In the follow, we recall the treatment exposed in [DFKP04].

3.1.1 Discrete and Continuous Multi-resolution Models

Discrete mesh-based multi-resolution 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 multi-resolution 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 multi-resolution models are used in a dynamic application, this difference may often cause unpleasant "popping" effects. In other words, discrete multi-resolution models do not provide any mechanism for smooth transition between different levels of detail.

Variable-resolution 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 a multi-resolution 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.

3.1.1.1 Regular Representations

Multi-resolution 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 semi-regular 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).

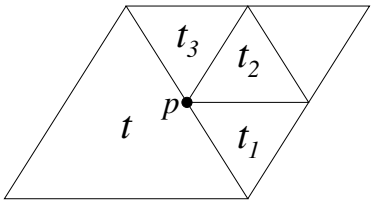


Figure 3.1: A non conforming mesh: triangles t_1, t_2 and t_3 intersect triangle t not at a common simplex [Dan05].

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 sub-triangles. 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 subdivision 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 3.1, 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,

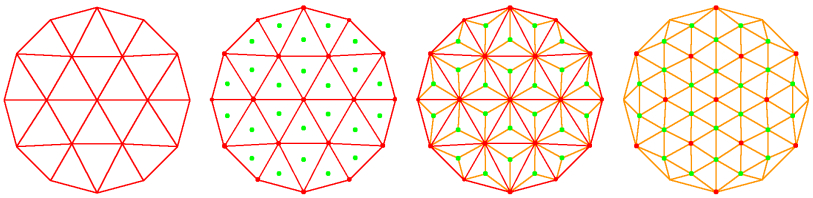


Figure 3.2: Mesh refinement by the $\sqrt{3}$-operator [DFKP04].

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 and all other vertices will keep their original valence (see Figure 3.2). 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 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 3.3) by the vertex v, called the *split vertex* of d, which

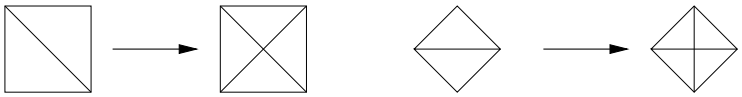


Figure 3.3: Split sets corresponding to the two types of diamonds generated by recursive triangle bisection

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 3.4.

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 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

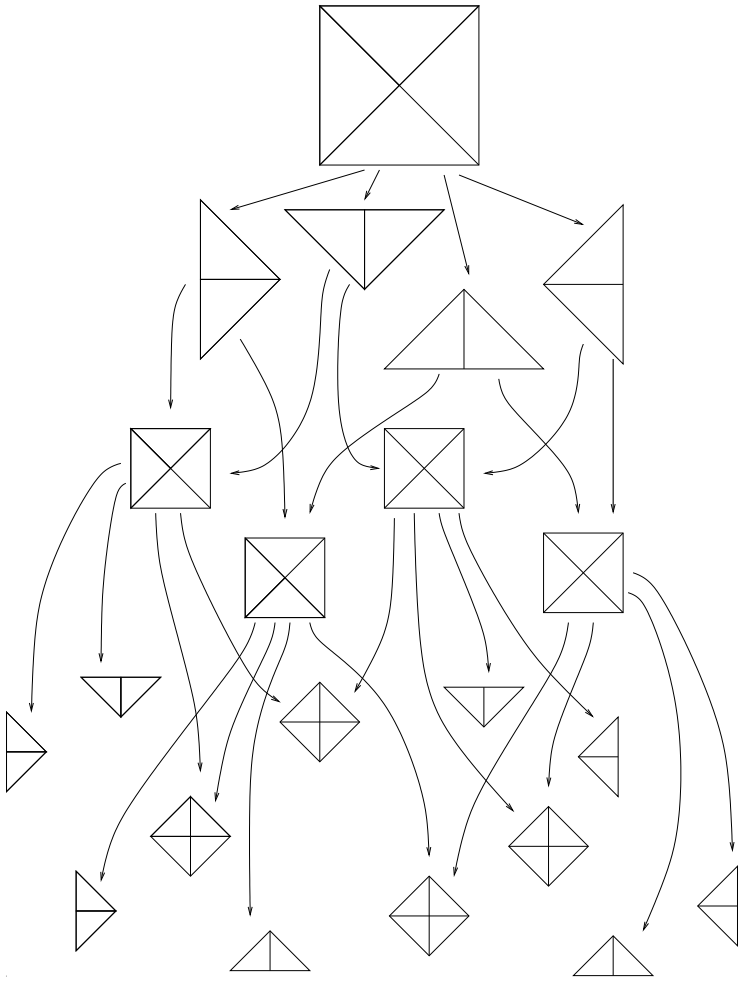


Figure 3.4: The DAG-based vertex dependencies, where each node represents both a vertex and its corresponding diamond (diamonds on the border contain just two triangles) [DFKP04].

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.

3.1.1.2 Irregular 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

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), has been described in Section 2.2, 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. For a complete overview about the researches that lead to the definition of the MT and the different application fields touched, see [DFM$^+$05a].

Generally speaking, triangular multi-resolution 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 multi-resolution models implement special instance of this framework, as shown in [DFM02]. Also the regular models we have reviewed in Subsection 3.1.1.1, can be viewed as specific instances of such framework.

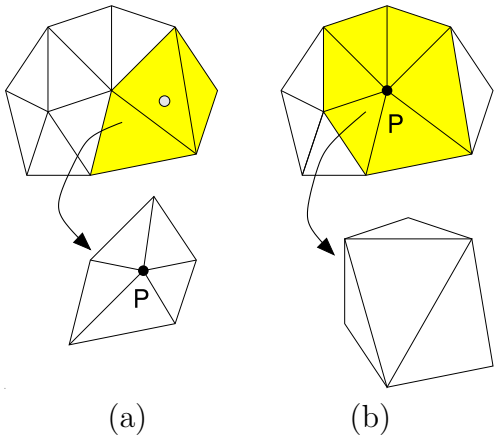


Figure 3.5: (a) Vertex insertion; (b) Vertex removal.

Compact data structures for multi-resolution 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 3.5a). 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 3.5b).

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 $j+1$ 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.

Data structures based on edge collapse/vertex split. 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 level of detail 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 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 3.6). 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

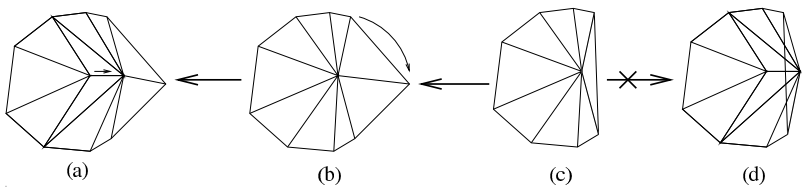


Figure 3.6: Foldover: original sequence of collapses is (a)-(b)-(c) and the reverse sequence of splits is correct; if the central vertex is split as in sequence (c)-(d) a foldover occurs [DFKP04].

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.

Multi-resolution 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.

3.1.2 Variable-resolution 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 [DDFMP01]).

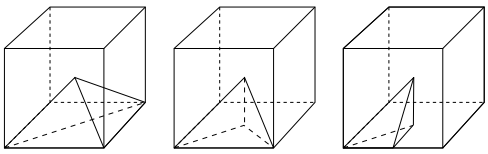


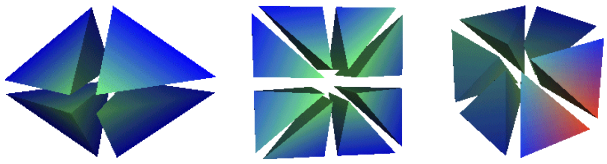
Figure 3.7: (a) Subdivision of the initial cubic domain into six tetrahedra. Examples of (b) a 1/2 pyramid, (c) a 1/4 pyramid, and (d) a 1/8 pyramid (see [LDFS01]).

3.1.2.1 Regular 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 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 3.7).

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



(a) (b) (c)

Figure 3.8: (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]).

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 3.8).

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 3.1.1.1). 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.

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 3.1.1.1. 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.

3.1.2.2 Irregular Representations

A multi-resolution 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]. Tetrahedron collapse to a new vertex has also been used to simplify tetrahedral meshes [CM02].

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.

In [CDFM+04], a 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]. 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.

3.2 Data Structures for Non-Manifold Modeling

The problem of representing and manipulating spatial objects with a complex topology and formed by parts of different dimensionalities has been faced in solid modeling long time ago (see, e.g., [GCP90, RO90, Wei88]) because of its relevance in CAD/CAM applications. Non-manifold modelers allow combinations of wire-frame, surface, solid and cellular decompositions of a 3D object to be represented and manipulated in a consistent topological representation. Complex non-manifold objects are described through meshes with a non-manifold and non-regular domain (i.e., with parts of different dimensionalities), encoded in non-manifold data structures.

Motivations for using non-manifold data structures have been pointed out by several authors [CA95, GCP90, RO90, Wei88]. For instance, Boolean operators are closed in the non-manifold domain, sweeping or offset operations may generate parts of different dimensionalities, non-manifold topologies are required in different product development phases, such as conceptual design, analysis or manufacturing [CA95, SWH95].

In this Section, we focus on data structures for two-dimensional complexes embedded in $\mathbb{E}^3$, designed specifically for representing non-manifold objects. We briefly describe three edge-based data structures: a specialization of the Partial Edge (PE) data structure [LL01] for cell 2-complexes embedded in $\mathbb{E}^3$; the Loop Edge-use (LE) data structure [McM00], which specializes the Radial Edge data structure proposed by Weiler [Wei88] for cell 2-complexes to regular simplicial 2-complexes, in which the star of a vertex consists of a single connected component; and the Directed Edge (DE) data structure [CKS98], which can encode any simplicial complex embedded in $\mathbb{E}^3$.

For a more detailed review on data structures for simplicial complexes, interested readers can refer to [DH05].

3.2.0.3 The Simplified Partial Entities data structure

The *Partial Entities (PE)* data structure [LL01] has been proposed for encoding cell complexes which define the boundary subdivision of 3D objects and has been specialized to two-dimensional simplicial complexes in [DH05].

The simplified PE data structure encodes: *edges* and *vertices*, plus two oriented entities, i.e., *faces* and *partial-edges*. Each face is bounded by one oriented loop, which consists of a circle of partial-edges. Each partial-edge corresponds to the appearance of an edge on the oriented loop bounding a face. Thus, if there are m faces incident at edge e, the PE data structure stores m partial-edges of e. A wire edge we has a loop that consists of two partial-edges of we. Here, we consider the simplified version of the original PE data structure, without the loops and entities like the shell and the region, which have been introduced in

the PE structure to encode cell complexes with several connected components.

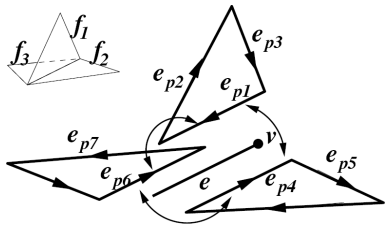


Figure 3.9: Example of the PE data structure: relations at non-manifold edge e shared by three faces: f_1, f_2 and f_3 [DH05]

The simplified PE data structure for 2D simplicial complexes encodes the following information.

- For each face f : a reference to a partial-edge on its boundary.
- For each edge e, a reference to a partial-edge that describes e.
- For each partial-edge e_p bounding face f , there is a reference to each of the following:
  - the corresponding unoriented edge e (e in the example of Figure 3.9);
  - the face f (f_1 in the example);
  - the previous adjacent partial-edge ordered in counterclockwise direction around e (e_p4 in the example);
  - the next adjacent partial-edge ordered around e (e_p6 in the example);
  - the previous partial-edge in counter-clockwise direction on the boundary of f (e_p3 in the example);
  - the next partial-edge on the boundary of f (e_p2 in the example);
  - the start vertex of e_p (v in the example);
- For each vertex v: the list of partial-edges e_p that start at v

3.2.0.4 The Loop Edge-use data structure

The *Loop Edge-use (LE)* data structure [McM00] simplifies the Radial Edge data structure [Wei88] developed for cell 2-complexes embedded in $\mathbb{E}^3$ to regular simplicial complexes in which non-manifold singularities occur only at edges. Thus, the LE data structure does not represent wire edges, and is based on the assumption that the link of each vertex

consists of a single connected component. In this representation, each 2-simplex has a single orientation.

The entities encoded by the LE data structure are: faces, edges, vertices and edge-uses, where an edge-use is similar to the partial-edge in the PE data structure and represents the association between an edge and a face incident at it.

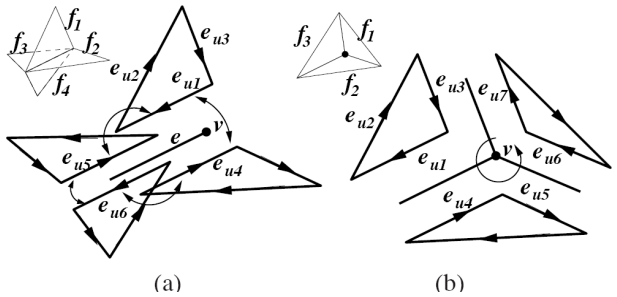


Figure 3.10: Elements of the LE data structure: (a) Edge-uses of a non-manifold edge e shared by four faces: f_1, f_2, f_3 and f_4; (b) Edge-uses starting at a vertex v [DH05]

The LE structure encodes the following information: (we refer to the examples in Figure 3.10 to illustrate it)

- For each face f, a reference to one edge-use on the boundary of f (In Figure 3.10(a), f_1 has a reference to e_u1);
- For each edge e, a reference to one edge-use that is associated with e (In Figure 3.10(a), e has a reference to e_u1);
- For each edge-use e_u on the boundary of face f , there is a reference to each of the following: (We illustrate the fields of e_u with e_u1 in Figure 3.10(a). For clarity, we illustrate the last of field of e_u with e_u1 in Figure 3.10(b).)
  - the face f (f_1 in the example);
  - the corresponding unoriented edge e (e in the example);
  - the next adjacent edge-use ordered around e in counter-clockwise direction (e_u5 in the example);
  - the edge-use following e_u in counter-clockwise direction on the boundary of f (e_u2 in the example);
  - the start vertex v of e_u (v in the example);
  - the next edge-use that starts at v (In the example of Figure 3.10(b), e_u1 has a reference to e_u5);

- For each vertex v, a reference to an edge-use that starts at v (In Figure 3.10(b), v has a reference to e_u1);

3.2.0.5 The Directed Edge data structure

The *Directed Edge (DE)* data structure [CKS98] is an extension of the *Half-Edge* data structure [Man88] proposed for two-dimensional cell complexes within a manifold domain.

It is based on the concept of a directed edge. A *directed edge* e_d of an edge e in a 2-complex is an occurrence of e on the boundary a triangle incident at e. A directed edge is similar to the edge-use in the LE data structure and to the partial-edge in the PE data structure. In the DE data structure, the entities are directed edges and vertices. Triangles and unoriented edges are not explicitly encoded. Triangles are implicitly encoded by the edges on their boundary. The association between a triangle and its three edges is through indexing. The i-th triangle, f_i is described by the $3i$-th, $(3i+1)$-th and $(3i+2)$-th directed edges, which form the oriented boundary of f_i. Wire edges are represented as directed edges.

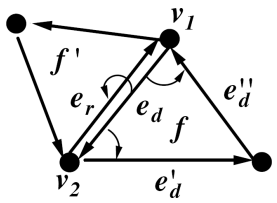


Figure 3.11: An illustration of the relations encoded at a directed edge e_d [DH05]

The DE data structure encodes the following information:

- For each triangle f , the three directed edges on the boundary of f ;
- For each directed edge e_d on the boundary of face f , there is a reference to each of the following: (see Figure 3.11 for the illustration of the symbols)
  - the start vertex v_1;
  - the end vertex v_2;
  - the next adjacent directed edge e_r ordered in counterclockwise direction around the unoriented edge between v_1 and v_2;
  - the previous directed edge $e_d^{''}$ bounding f in counterclockwise order;
  - the next directed edge $e_d^{'}$ bounding f in counterclockwise order;
- For each vertex v, one directed edge from each connected component of the link of v.

3.3 Multi-resolution Modeling over the Network

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. According with the subdivision stated in Section 3.1, progressive models are continuous LOD models, since they have a fine granularity, but they do not allow extracting meshes in which the resolution varies in different parts of the domain. In Subsection 3.3.2, we review proposals for a client/server approach to multi-resolution modeling. They have been specifically designed for streaming and visualization purposes.

3.3.1 Progressive Models

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 higher 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 [Hop98, 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 [AD01], 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 triangle.

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 split at each step). Experimental results report an average cost of 3 bits per tetrahedron.

Bischoff and Kobbelt [BK02] propose algorithms for the robust transmission of geometric three dimensional data, providing a technique that progressively sends part of the geometry and allows recovering parts eventually lost. In this case, the system provides an approximation of the original model based on reconstructing the original mesh according with the topology streamed in a secure channel.

3.3.2 Client/Server Approaches

In [CDFM+00], De Floriani et al. have developed a data structure for the 2D Multi-Tessellation in a client-server architecture. They have developed an application to selective transmission of surface meshes,which is built on top of a compact data structure for two-dimensional MT, generated through vertex insertion/removal operations. This application allows the client to make LOD queries and obtain instructions to selectively refine meshes on-the-fly, without the need of receiving and storing the whole LOD model locally. Only the server stores the MT data structure and contains a mechanism for extracting selectively refined meshes from it. A client sends requests to the server by specifying the required LOD and the size of the mesh that can be managed locally. The server selectively refines a mesh according to client requests, and transmits it progressively to the client. The basic issue here is reducing communication time and, thus, the amount of information sent from the server to the client to describe the extracted mesh. Efficiency is achieved by a compact encoding of mesh updates, a suitable mechanism of translation tables, which is necessary to keep consistency in data structures at the client and at the server, and a caching mechanism at the client. This approach is based on the assumption that the server has good storing and computation power, most of all for the generalization to a multiple clients system.

El-Sana and Sokolovsky [ESS03] propose a client/server application based on the view-dependent tree [ESV99], with cache on the client and a prediction mechanism to adapt more quickly the visualization and reduce the transmission time. The idea is to subdivide in blocks the view-dependent tree that allows the selective refinement and to maintain on the server side a list of active nodes for every client connected: at every request from a client, the server, according to the active node list, sends to the client the update operations needed to satisfy the visual query. They propose also different solutions from the Local Area Network and the Wide Area Network, considering the different transmission times. Also in this approach, the idea is the the server is a capable computer, with a lot of storing space available and a good computation power.

Kobbelt et al. [KLK04] propose a client/server approach for view dependent streaming of Progressive Meshes: the server store a Progressive Mesh representation and a vertex hierarchy built on the basis of view-dependent criteria. This approach differs from the standard PM streaming, in which the transmission of the sequence of updates is guided typically from geometric constraints, because the streaming is guided from visual criteria: the server sends to the client the base mesh and part of the vertex hierarchy, and, according to truly selective refinement operations generated from the client requests, it sends the rest of the structure, until the streaming is completed. This approach suffers of the same limitation of all the progressive models, i.e. the impossibility of extracting mesh with variable resolution in different parts of the domain, but is interesting for the fact that the streaming process is, in part, guided from view-dependent criteria.

Chapter 4

The Non-manifold Multi-Tesselation

In this Chapter we define *Non-manifold Multi-Tessellation* (NMT), a general model for representing at different Levels-Of-Detail (LOD) non-manifold objects described by two dimensional simplicial complexes embedded in 3D space, as proposed in [DFMPS02] and described in detail in [DFMPS04]. NMT is the natural extension to the non-manifold domain of the Multi-Tesselation model described in Section 2.2. We focus on a specific instance of an NMT built on the basis of the most common operation for non-manifold mesh simplification, i.e., vertex-pair contraction.

The remainder of the Chapter is organized as follows. In Section 4.1 we introduce some useful background notions. In Section 4.2, we define a new concise topological data structure for describing a 2D non-manifold simplicial complex. In Section 4.3, we define the non-manifold MT in the general case. In Section 4.3.2, we discuss the selective refinement query, and we introduce the primitives needed for answering such query. In Section 4.4, we introduce a compact data structure for a non-manifold MT built through vertex-pair contraction, and we evaluate its space requirements. In Section 4.5, we describe how mesh update primitives, that are involved in selective refinement, can be implemented on this data structure. Section 4.6 contains some preliminary experimental results.

4.1 Triangle-Segment Meshes

In this section, we give an informal definition of triangle-segment meshes, that we use for defining a Non-manifold Multi-Tessellation.

We call a *triangle-segment mesh* a two-dimensional simplicial complex, i.e., a finite set Σ of vertices, edges and triangles in $\mathbb{E}^3$, with the following properties:

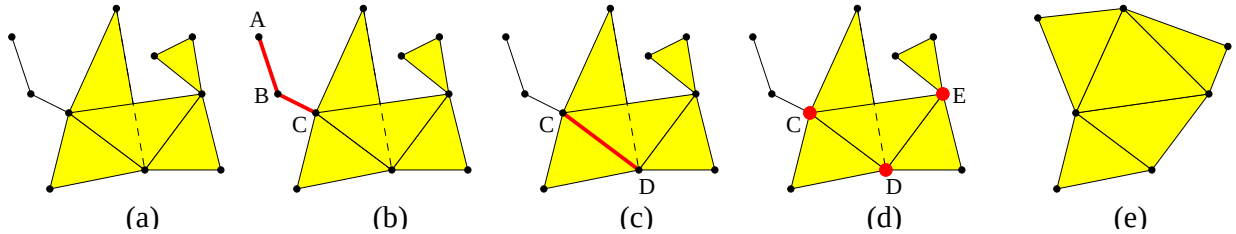


Figure 4.1: (a) A triangle-segment mesh; (b) (A, B) and (B, C) are wire-edges; (c) (C, D) is a non-manifold edge; (d) C, D, and E are non-manifold vertices. (e) A manifold triangle-segment mesh.

- For any triangle t, the three edges of t must be in Σ;
- For each edge e, the two vertices of e must be in Σ;
- Two triangles t_1 and t_2 of Σ cannot intersect properly, but they can share either an edge or a vertex;
- Two edges e_1 and e_2 of Σ cannot intersect properly, but they can share a vertex;
- A triangle t and an edge e cannot intersect properly, but e can be an edge of t, or e and t can share a vertex.

For the sake of brevity, in the remainder of this Chapter, a triangle-segment mesh will be called simply a *mesh*. Moreover, we restrict our attention to connected meshes. Some examples of meshes are shown in Figure 4.1.

An edge e of a mesh Σ is called a *wire-edge* if no triangle $t \in \Sigma$ exists such that e is an edge of t, i.e., if e is a top-edge (see Figure 4.1b). Otherwise, e is called a *triangle-edge*. A mesh is completely described by the set of its triangles plus the set of its wire-edges. We define the *size* of a mesh as the number of its triangles plus the number of its wire-edges.

A triangle-edge is a *manifold* edge if it has one or two incident triangles; if it has three or more incident triangles, then it is a *non-manifold* edge (see Figure 4.1c). A vertex v is a *manifold* vertex if one of the following conditions holds: at most two wire-edges are incident at v, and no triangle is incident at v; or no wire-edge is incident in v, and its incident triangles form a single fan, which may be either open or closed (this implies that all edges incident at v are manifold). Otherwise it is a *non-manifold* vertex (see Figure 4.1d).

A *manifold triangle mesh* is a mesh in which there are no wire-edges, and all edges and vertices are manifold (see Figure 4.1e).

We consider the following parameters for a mesh Σ:

- n denotes the number of vertices,

Σ	triangle-segment mesh
n	number of vertices
m	number of triangles
l	number of wire edges
c	number of edge-connected components of triangles incident at non-manifold vertices
a	number of triangles incident at non-manifold edges
$\mathcal{M}$	Non-manifold Multi Tessellation
Σ_0	base mesh of an NMT (lowest resolution)
Σ_h	reference mesh of an NMT (highest resolution)
M^+	refinement modification
M^-	coarsening modification

Table 4.1: Symbols that refer to meshes and NMTs.

- l denotes the number of wire-edges,
- m denotes the number of triangles of Σ,
- c denotes the sum, over all non-manifold vertices v of Σ, of the number of edge-connected components formed by the triangles incident in v,
- a denotes the sum, over all non-manifold edges e of Σ, of the number of triangles incident on e.

Quantities l, c and a give a measure of the degree of "non-manifoldness" of mesh Σ. In a manifold triangle mesh, $l = c = a = 0$, and $m < 2n$. Usually, meshes that are encountered in the applications are "nearly" manifold, i.e., $l, c, a << m$, and $m \approx 2n$ (this latter equality being a consequence of the Euler formula). Table 4.1 summarizes the meaning of symbols defined above, as well as the meaning of other symbols defined in Section 4.3.

Throughout the chapter, in analyzing the complexity of data structures, we will make the assumption that both numbers (either integer or real number), and indices (pointers) require four bytes each to be stored.

4.2 A Data Structure for Triangle-Segment Meshes

In this section, we describe the data structure we have designed for encoding a triangle-segment mesh as defined in Section 4.1.

This data structure have been fully implemented in a C++ language library, called *TSM-lib*, which is available in the software repository of the European Network of Excellence AIM@SHAPE.

We encode explicitly the *vertices* (V) and the *triangles* (T) of a mesh Σ. Among these two types of entities, we define the following four relations:

- *Vertex-Vertex* (VV) relation: it associates a vertex v with all its adjacent vertices. For clarity, we split such relation into two parts:
  - relation VV^{wir}, which contains those vertices adjacent to v through a wire-edge;
  - relation VV^{trg}, which contains those vertices adjacent to v through a triangle-edge.

  Both VV^{wir} and VV^{trg} are symmetric relations: $v' \in \mathrm{VV}^{wir}(v)$ if and only if $v \in \mathrm{VV}^{wir}(v')$, and the same holds for VV^{trg}.
- *Vertex-Triangle* (VT) relation: it associates a vertex v with all its incident triangles.
- *Triangle-Vertex* (TV) relation: it associates a triangle t with its three vertices v_1, v_2, v_3.

  Relation VT and TV are mutually inverse: $t \in \mathrm{VT}(v)$ if and only if $v \in TV(t)$.
- *Triangle-Triangle* (TT) relation: it associates a triangle t with three pairs of triangles, one for each edge e of t. Let v and w be the two endpoints of e. Then relation $\mathrm{TT}(t, v, w) = (t_1, t_2)$, where t_1 is the first triangle incident at edge e, that is encountered from t in counterclockwise order, and t_2 is the first one that is encountered from t in clockwise order, considering edge e as oriented from v to w (see Figure 4.2). If t has no adjacent triangles along edge e, then $t_1 = t_2 = t$.

  Relation TT is symmetric at each edge, in the sense that $\mathrm{TT}(t, v, w) = (t_1, t_2)$ if and only $\mathrm{TT}(t_1, v, w) = (\ldots, t)$ and $\mathrm{TT}(t_2, v, w) = (t, \ldots)$. Moreover, $\mathrm{TT}(t, v, w) = (t_1, t_2)$ if and only if $\mathrm{TT}(t, w, v) = (t_2, t_1)$.

Relation TT implicitly provides a cyclic list of the triangles incident at an edge e, since it allows moving from one triangle to the next one according to either a clockwise, or a counterclockwise order.

If we consider the set $\mathrm{VT}(v)$ of triangles incident in a vertex v, and the adjacency relations existing among the triangles of $\mathrm{VT}(v)$ along the edges incident in v, then set $\mathrm{VT}(v)$ is partitioned into one or more edge-connected components. For instance, for vertex E in Figure 4.1d, $\mathrm{VT}(D)$ consists of two edge-connected components.

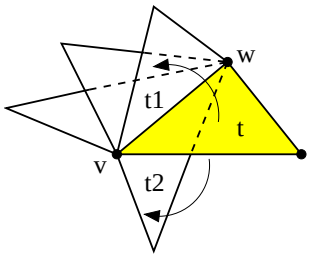


Figure 4.2: Relation $\mathrm{TT}(t, v, w)$ for a triangle t and edge $e = (v, w)$ of t: the triangles belonging to the relation are t_1 and t_2.

We define a partial version of relation VT, denoted with VT^*, that associates a vertex v with just one triangle for each edge-connected component of $\mathrm{VT}(v)$. Note that in a triangle-segment mesh, vertex-based relations can involve an arbitrary number of elements, while triangle-based relations involve a constant number of elements.

We have designed our data structure for triangle-segment meshes with a negligible overhead with respect to standard data structures for manifold triangle meshes, when it is used to represent a manifold triangle mesh. In a manifold triangle mesh, for each vertex v, relation $\mathrm{VV}^{wir}(v)$ is empty, and relation $\mathrm{VT}^*(v)$ contains just one triangle. Moreover, for each triangle t and edge $e = (v, w)$ of t, the two triangles provided by relation $\mathrm{TT}(t, v, w)$ are coincident.

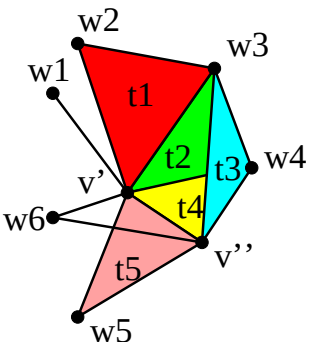
For each vertex v, the data structure stores:

- The three coordinates of v;
- If $\mathrm{VV}^{wir}(v)$ is not empty, a link to every vertex w such that $w \in \mathrm{VV}^{wir}(v)$;
- A link to one triangle incident in v for each connected component of $\mathrm{VT}(v)$, i.e., relation $\mathrm{VT}^*(v)$. The representative triangle in VT^* for a connected component is selected arbitrarily.

For each triangle t, the data structure stores:

- Links to the three vertices v_1, v_2, v_3 of t, i.e., relation $\mathrm{TV}(t)$;
- For each edge $e = v_i v_{i+1}$ of t,
  - if e is non-manifold, links to the two triangles in $\mathrm{TT}(t, v_i, v_{i+1})$;
  - if e is manifold, link to one of such triangles (since they are the same).

Figure 4.3 shows an example of the data structure. Note that, if $\mathrm{TT}(t, v_i, v_{i+1})$ is stored, then $\mathrm{TT}(t, v_{i+1}, v_i)$ is also virtually stored (it is sufficient to reverse the triangle pair).



vertex	VV^{wir}	VT^*
w_1	v'	–
w_2	–	t_1
w_3	–	t_2
w_4	–	t_2
w_5	–	t_5
w_6	v', v''	–
v'	w_1, w_6	t_4
v''	w_6	t_5

triangle	TV	TT
t_1	v', w_3, w_2	t_2, t_1, t_1
t_2	v', w_4, w_3	t_4, t_3, t_1
t_3	v'', w_4, w_3	t_4, t_2, t_3
t_4	v'', w_4, v'	t_3, t_2, t_5
t_5	w_5, v', v''	t_5, t_4, t_5

Figure 4.3: A triangle-segment mesh, and the tabular representation of the data structure encoding it.

4.2.1 Retrieval of Non-Stored Relations

Information stored in this data structure allow retrieving all other vertex-based and triangle-based relations with a time complexity linear in the output size.

In order to implement the basic operations *Mesh_Refinement* and *Mesh_Coarsening* for selective refinement, as explained in Section , we need retrieving relations VV^{trg} and VT for a given vertex v, and retrieving the cycle of triangles incident in an edge e, given a triangle incident in e.

In the following, we present the algorithms for these three operations, all working in linear time in their output size.

4.2.1.0.1 Relation VT**.** We describe the algorithm that, starting from a triangle $t \in VT^*(v)$, finds all triangles of $VT(v)$, which belong to the same edge-connected component as t. We retrieve the complete VT relation by repeating this process for all triangles in $VT^*(v)$. Given t, the algorithm works as follows:

1. Output triangle t, and mark t;
2. Let v_1, v_2 be the two vertices of t different from v, and let t_1, t_2, t_3, t_4 be the four triangles stored in relations $TT(t, v_1, v)$ and $TT(t, v_2, v)$;
3. For each $i \in [1, 4]$, if t_i is not marked, then recursively call the algorithm starting from t_i.

At the end of the process, we clear the marks for all output triangles.

4.2.1.0.2 Relation VV^{trg}. For a vertex v, relation $VV^{trg}(v)$ can be constructed with the same approach described to retrieve relation VT. It is sufficient to output, for each triangle of $t \in VT(v)$, the two vertices of t different from v. Since the same vertex may appear in many triangles, to prevent duplicates we mark each vertex as we output it, and we output just vertices that have not yet been marked.

4.2.1.0.3 The cycle of triangles around an edge. Let t be a triangle and $e = (v, w)$ be an edge of t. The following algorithm finds the list of all triangles incident in e, sorted in counterclockwise order around e if e is considered as oriented from v to w:

1. Output t;
2. Let t_1, t_2 be the two triangles stored in $TT(t, v, w)$;
3. Repeat the same process, from step 1, starting from t_1;

The process stops when triangle t is encountered again.

4.2.2 Implementation and Space Requirements

Before evaluating the space complexity of the data structure, we need to give more details about its implementation. One goal of our implementation is that the structure should scale well with respect to the degree of non-manifoldness of a mesh.

We take as a reference a standard topological data structure for manifold triangle meshes, which stores vertex coordinates and relations VT^*, TV, and TT. In the manifold case, relation VT^* contains just one triangle and relation TT contains only three triangles (i.e., one per edge of a given triangle). Because all stored relations are constant, such data structure can be implemented with arrays, and requires $3n$ real numbers for vertex coordinates, n indexes for relation VT^*, $3m < 6n$ indexes for relations TV, and TT. Thus, the overall space complexity is equal to $12n + 4n + 24n + 24n = 64n$ bytes.

In the following, we present an implementation of our non-manifold data structure, that introduces a negligible overhead with respect to the above manifold data structure, when the number of non-manifold elements in a mesh is small.

We maintain an array containing all vertices and an array containing all triangles. We use garbage collection on such arrays in order to support the dynamic creation and deletion of triangles and vertices during a selective refinement algorithm. For each vertex v, we store:

- the three coordinates of v, requiring 12 bytes.

- a flag (on two bits) containing:
  - 0 if v is manifold and just triangles are incident in v (no wire-edges);
  - 1 if just wire-edges are incident in v (no triangles);
  - 2 if v is non-manifold and just triangles are incident in v (no wire-edges);
  - 3 if both triangles and wire-edges are incident in v.
- an index (on 4 bytes) with the following meaning, depending on the flag:
  - if 0: pointer to the unique triangle in $\mathrm{VT}^*(v)$;
  - if 1: pointer to a linked list containing $\mathrm{VV}^{wir}(v)$;
  - if 2: pointer to a linked list containing $\mathrm{VT}^*(v)$;
  - if 3: pointer to the first element of an array of two linked lists containing $\mathrm{VV}^{wir}(v)$ and $\mathrm{VT}^*(v)$, respectively; the same pointer shifted by one provides the address of the second element of the array.

  Each element of list $\mathrm{VV}^{wir}(v)$ is a pair containing the index of one vertex w and a pointer to the position of v in $\mathrm{VV}^{wir}(w)$. This latter pointer is not necessary to support mesh traversal, but it is used to improve the performance of algorithms that update the data structure, which will be described in Section 4.5.

For each triangle t, we store:

- three vertex indexes for relation $\mathrm{TV}(t)$, requiring 12 bytes.
- three flags (on 1 bit each), one for each edge $e = v_i v_{i+1}$ of t. Each flag is set to 0 if e is manifold, and to 1 otherwise.
- three indexes (on 4 bytes each), one for each edge e of t, with the following meaning, depending on the flag associated with e:
  - if 0: pointer to one triangle, corresponding to $\mathrm{TT}(t, v_i, v_{i+1})$, as in the case of manifold triangle meshes;
  - if 1: pointer to the first element of an array of two triangles containing relation $\mathrm{TT}(t, v_i, v_{i+1})$; the same pointer shifted by one provides the address of the second element of the array.

Linked lists have been used in order to allow performing vertex-pair contractions and vertex expansions on-line on the data structure. However, such lists are on average composed of very few elements (in our experiments, the average length is very close to one, and

the maximum length never exceeds four). Thus, the computational overhead involved by managing linked lists is negligible.

The overall space required by the vertex array is $16n$ bytes $+2n$ bits. The sum of the cardinalities of all lists (stored separately) for relation VV^{wir} is equal to $2l$ because each wire-edge $e = vw$ contributes with one element both to $\mathrm{VV}^{wir}(v)$ and $\mathrm{VV}^{wir}(w)$. The sum of the cardinalities of all lists (stored separately) representing relation VT^* is equal to c. Thus, the overall space required by all linked lists referenced by vertices is $6l + 2c$ pointers, i.e., $24l + 8c$ bytes.ff

The overall space required by the triangle array is $24m$ bytes and $3m$ bits. The total size of all arrays representing relation TT at non-manifold edges is equal to $2a$ since each triangle t incident in a non-manifold edge e appears in the TT relation of two triangles. Since the space for the first element of the array has already been taken into account, we have to add just the space needed for the second element, i.e., overall a pointers.

Thus, the total space complexity of the data structure is equal to $16n + 24m + 24l + 8c + 4a$ bytes and $2n+3m$ bits. If the mesh is manifold, this reduces to $16n+24m \simeq 16n+48n = 64n$ bytes and $2n + 6n = 8n$ bits $= n$ bytes, i.e., $65n$ bytes in total. Thus, the overhead introduced by the above data structure, when it is used to encode a manifold triangle mesh, is just one byte per vertex. Our data structure is more compact than other structures proposed for non-manifold meshes: its storage cost is about one order of magnitude lower than the cost of the radial-edge data structure; one fourth of the cost of the directed-edge data structure; and about half the cost of the incidence graph. The main reason is that edges are not represented, and each other entity is represented only once. Our data structure supports traversal algorithm for evaluating vertex- and triangle-based relations that are more efficient than those supported by the incident graph, and as efficient as those supported by the other data structures. The main drawback of our data structure is that it cannot support attribute information associated with edges, since edges are just implicitly represented. For the same reason, traversal algorithms for evaluating edge-based relations are suboptimal, since they require traversing the stars of endpoints.

4.3 A Multi-resolution Model for Triangle-Segment Meshes

In this section, first we describe the Non-manifold Multi-Tessellation (NMT), which is a natural extension to the non-manifold domain of the model developed for the manifold case in [Pup98] and in subsequent work. Next, we describe the basic structure of an algorithm that performs *selective refinement* on an NMT, which is also a natural extension of an algorithm presented in [DFMMP00].

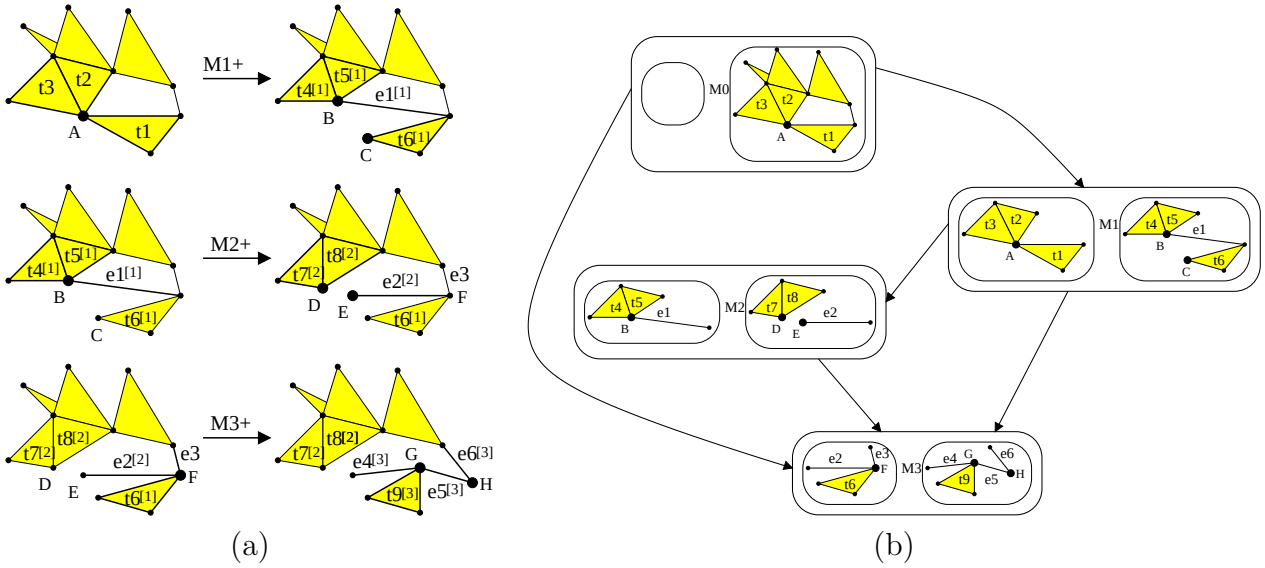


Figure 4.4: (a) A mesh refined by a sequence of three modifications. The elements (triangles and wire-edges) involved in each modification are drawn with thick edges. The elements created by a modification are labeled with its number between square brackets. (b) The corresponding NMT DAG. Each node contains the pair of sets of cells that defines the corresponding refinement and coarsening modification.

4.3.1 Non-Manifold Multi-Tessellation

First of all, for clarity, we recall the fundamental notions of the Multi-Tessellation model described in 2.2, which, in the follow, we extends to the non-manifold case.

The basic ingredient in the NMT is the concept of modification. A *modification* of a mesh Σ is an operation that deletes some set of cells (vertices, edges, and/or triangles) from Σ, and replaces them with another set of cells, under the constraint that the result must be a mesh Σ'. A modification is completely described by the pair (C, C'), where C is the set of cells (triangles and wire-edges) of Σ which are not in Σ' (this may include wire-edges of Σ that are triangle-edges in Σ'), and, symmetrically, C' is the set of cells of Σ' that are not in Σ.

We focus on modifications that change the size of a mesh by either increasing it (*refinement*), or decreasing it (*coarsening*). We use the notation M^+ for a refinement modification, and M^- for its inverse, i.e., a coarsening modification. Examples of refinement modifications are shown in Figure 4.4a.

Let $M_1^+, M_2^+, \ldots, M_h^+$ be a sequence of modifications that progressively refine a coarse mesh Σ_0 into a mesh Σ_h at full resolution (see Figure 4.4a). Note that the reversed sequence $M_h^-, \ldots, M_2^-, M_1^-$ progressively coarsens Σ_h into Σ_0. A multi-resolution mesh encompasses

all meshes that can be constructed from such modifications. This is possible by defining a *partial order* that describes *mutual dependencies* between pairs of modifications.

A *Non-manifold Multi-Tessellation* is a partially ordered set of *nodes* $\mathcal{M} = (\{M_0, M_1, \ldots, M_h\}, \prec)$, where each node M_i represents both a refinement modification M_i^+ and its inverse coarsening modification M_i^-. By convention, node M_0 consists of a pair $(\emptyset, \Sigma_0)$, where Σ_0 is a (coarse) mesh, called the *base mesh* of $\mathcal{M}$.

The partial order $\prec$ describes the following *mutual dependencies* between pairs of modifications, as in a manifold MT, having M_0 as minimum element:

1. $M_i \prec M_j$ if, considering the associated refinement modifications $M_i^+ = (C_i, C_i')$ and $M_j^+ = (C_j, C_j')$, we have that $C_i' \cap C_j \neq \emptyset$.
2. if $M_i \prec M_j$ and $M_j \prec M_k$, then $M_i \prec M_k$ (transitive closure).

The intuition behind condition (1) is that, if some element introduced by M_i^+ is deleted by M_j^+, then M_j^+ needs M_i^+ to be performed first. Symmetrically, M_i^- needs M_j^- to be performed first.

The partial order can be described as a Directed Acyclic Graph (DAG), where arcs connect pair of nodes satisfying condition (1) above. By convention, if $M_i \prec M_j$, then the corresponding arc is directed from M_i to M_j (see Figure 4.4b). Mesh Σ_h obtained by applying all modifications, in any total order consistent with $\prec$, is called the *reference mesh,* and it is the mesh at the highest possible resolution that can be obtained from $\mathcal{M}$.

A subset S of the nodes of an NMT is called *closed* with respect to the partial order if, for each node $M_j \in S$, all nodes M_i, such that $M_i \prec M_j$, are also in S. The refinement modifications corresponding to a closed subset of nodes can be applied to the base mesh Σ_0 in any total order extending $\prec$. This produces an *extracted mesh* Σ_S at a level of resolution intermediate between Σ_0 and Σ_h.

4.3.2 Selective Refinement on an NMT

As in the standard MT, a *selective refinement* query on a NMT consists of extracting a mesh, which satisfies some application-dependent requirements, such as approximating a spatial object with a certain accuracy which can be either uniform, or variable in space.

In order to formalize the parameters of a selective refinement query in an application-independent way, we consider a Boolean function τ, defined over the nodes of an NMT as follows: $\tau(M) = \mathit{true}$ if modification M^+ is not necessary in order to achieve the desired resolution, i.e., if $M^+ = (C, C')$, where the resolution of C is sufficient; $\tau(M) = \mathit{false}$ otherwise (i.e., the resolution of C is not sufficient.

The solution of a *selective refinement query* is the extracted mesh Σ_S, associated with the smallest closed set S such that for each node $M \notin S$, such that modification $M^+ = (C, C')$ acts on Σ_S (i.e., $\Sigma_S \cap C \neq \emptyset$), we have $\tau(M) = \mathit{true}$.

A simple example of selective refinement is spatial filtering: in this case, $\tau(M) = \mathit{false}$ if and only if M falls inside a region of interest. This means that all portion of the mesh falling inside such region should be expanded at the highest possible resolution, while the rest of the mesh can be at an arbitrarily low resolution.

Selective refinement can be performed by traversing the DAG describing an NMT and constructing a closed subset S of nodes, and its associated mesh Σ_S. An incremental algorithm was proposed in [DFMMP00] for the manifold case, which finds a solution to a new query by starting from the solution to a previous query. This same algorithm can be applied also to an NMT, upon implementation of suitable data structures and related methods.

The algorithm maintains a current closed set S of modifications, and a corresponding current mesh Σ_S. S and Σ_S are inherited from the last query, and they are updated based on a new function τ. Set S is updated through insertion and removal of nodes, and mesh Σ_S is updated by doing and undoing the corresponding modifications. Arcs connecting a node $M_1 \in S$ to a node $M_2 \notin S$ define a current *front* used to add/remove nodes to/from S:

- A node M which is the destination of a front arc is added to S (M^+ is done on Σ_S) if $\tau(M) = \mathit{false}$ (i.e., if M is necessary to achieve the desired resolution). In order to maintain S a closed set, every modification M', such that $M' \notin S$, and $M' \prec M$, is recursively added to S before adding M.
- A node M which is the source of a front arc is deleted from S if it $\tau(M) = \mathit{true}$, (i.e., if M^+ is not necessary). However, node M cannot be deleted if S contains some modification which depends on M. In this case, set $S \setminus M$ would not be closed.

A working example is shown in Figure 4.5.

The following basic primitives are needed in order to implement a selective refinement algorithm:

1. *Insertion_Test*: decide if a node $M \notin S$ can be added to S while maintaining S a closed set, i.e., if all nodes preceding M in the dependency relation are in S;
2. *Removal_Test*: decide if a node $M \in S$ can be removed from S while maintaining S a closed set, i.e., if all nodes that depend on M are not in S;

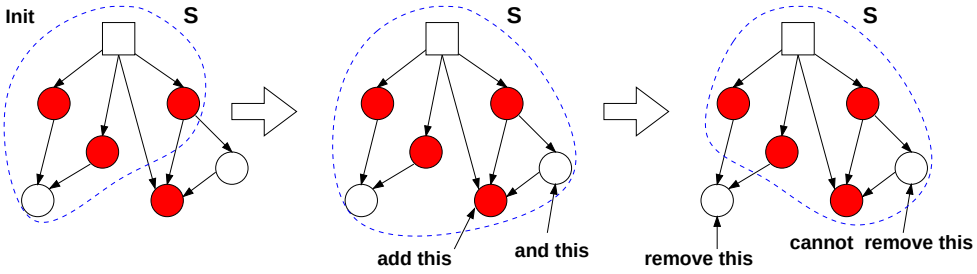


Figure 4.5: Selective refinement with an incremental approach. The square denotes the node M_0 corresponding to the base mesh. White nodes satisfy the function τ, dark nodes do not. The dashed line encloses the current set S.

3. *Dependencies_Retrieval*: retrieve all modifications which must be added to S in order to make the insertion of M in S feasible (i.e., all ancestors of M, which are not in S);

4. *Mesh_Refinement*: refine mesh Σ_S by applying M^+;

5. *Mesh_Coarsening*: coarsen mesh Σ_S by applying M^-.

Therefore, a data structure supporting such primitives is necessary in order to implement selective refinement efficiently on the NMT.

4.4 A Compact Data Structure for an NMT

We consider now an NMT constructed by progressively coarsening the reference mesh Σ_h through a sequence of vertex-pair contractions [ESV99, PH97]. A *vertex-pair contraction* M^- identifies two vertices v'_M and v''_M of a mesh (possibly connected by an edge e_M) into a unique vertex v_M. All edges and triangles incident at v'_M and/or in v''_M become incident into v_M. Triangles incident at edge e_M (if any) degenerate to edges. The inverse modification M^+ of a vertex-pair contraction M^- is called *vertex expansion*.

The modifications M_i^- shown in Figure 4.4 are vertex-pair contractions: M_3^-, M_2^-, M_1^- contract vertices G, H to F; D, E to B; and B, C to A, respectively.

A compact data structure for such an NMT is obtained by encoding information necessary to support contraction and expansion operations in the nodes, and by encoding implicitly the dependency among nodes through a binary tree.

4.4.1 Encoding Modifications

The root of the NMT, M_0, is encoded by a topological data structure representing the base mesh Σ_0, which will be described in the Section 4.2. Every other modification represents a vertex-pair contraction, and its inverse vertex expansion.

Let M be a node such that modification M^- is a vertex-pair contraction, and modification M^+ is a vertex expansion. Following Popovic and Hoppe [PH97], we represent a node M by storing:

- Offset vectors to retrieve the coordinates of v'_M and v''_M from the coordinates of v_M, and vice versa. In general, we need two offset vectors $v'_M - v_M$ and $v''_M - v_M$. If v_M is chosen to be the midpoint of segment $v'_M v''_M$, one offset vector is sufficient.
- One bit EDGEFLAG to indicate whether an edge e_M connecting v'_M and v''_M exists.
- For each edge and triangle incident at v_M, two bits SPLITFLAG, to indicate how the edge or triangle is modified when expanding v_M (see Figure 4.6):
  - 0 if the edge or triangle becomes incident in v'_M, and not with v''_M (see Figure 4.6a);
  - 1 if the edge or triangle becomes incident in v''_M, and not with v'_M (see Figure 4.6b);
  - 2 if the edge or triangle is split into two edges or triangles, one incident in v'_M and the other one in v''_M (see Figure 4.6c);
  - 3 (only possible for edges, if EDGEFLAG= 1), if the edge is split into two edges and a new triangles is created which is incident in both v'_M and v''_M (see Figure 4.6d).

Since our data structure for the current mesh will not store the edges explicitly, we associate the SPLITFLAG of an edge e with the vertex which is the endpoint of e different from v_M.

Figure 4.7 shows the use of the SPLITFLAGs in a vertex expansion, according to the convention. Vertices w_1, w_2 have SPLITFLAG= 0, thus edges (w_1, v) and (w_2, v) become (w_1, v') and (w_2, v'), respectively. Vertices w_3, w_6 have SPLITFLAG= 2, thus edges (w_3, v) and (w_6, v) become a pair of edges (w_3, v'), (w_3, v'') and (w_6, v'), (w_6, v''), respectively. Vertices w_4, w_5 have SPLITFLAG= 3, thus edges (w_4, v) and (w_5, v) become triangles $t_4 = (w_4, v', v'')$ and $t_5 = (w_5, v', v'')$, respectively. Triangle $t_1 = (w_2, w_3, v)$ has SPLITFLAG= 0, thus it becomes triangle (w_2, w_3, v'). Triangle $t = (w_2, w_3, v)$ has SPLITFLAG= 2, thus it becomes two triangles $t_2 = (w_2, w_3, v')$ and $t_3 = (w_2, w_3, v'')$.

The offset vectors for the coordinates are three real-valued numbers, requiring 12 bytes. The SPLITFLAGs are maintained in an array with the same sorting conventions as in [PH97]

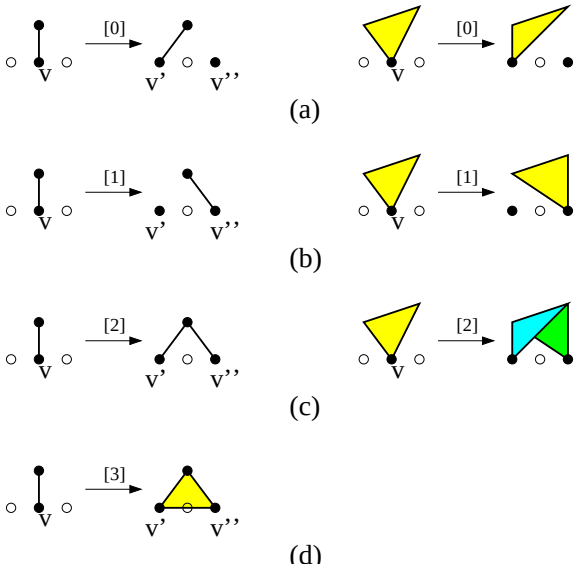


Figure 4.6: SPLITFLAG for an edge e and for a triangle t.

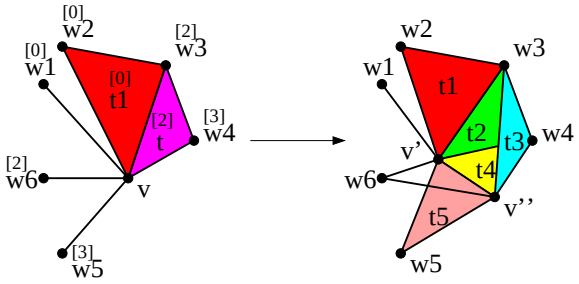


Figure 4.7: An example of using SPLITFLAGs in a vertex expansion. See text for explanation.

(the edges occupy the first positions, followed by the triangles). Each vertex in an NMT is labeled with a unique integer number (as explained in Section 4.4.2), and edges and triangles in the array are sorted lexicographically based on the labels of their vertices.

For a modification M, the size of the array is $2\,\text{degree}(v_M)$ bits, where $\text{degree}(v_M)$ is the total number of edges and triangles incident in vertex v_M. The length of the array does not need to be stored since $\text{degree}(v_M)$ can be found by looking at the configuration of the current mesh.

The total space complexity for storing all h modifications is thus equal to $12h$ bytes for offset vectors, and $(1 + 2D)h$ bits for the EDGEFLAGs and the SPLITFLAGs, where D is the average value of $\deg(v_M)$. The number h of modifications is bounded from above by the number n of vertices in the reference mesh Σ_h. The total cost for representing

modifications is equal to $12n + 8n = 20n$, by assuming for D a (pessimistic) upper bound of 31.5. The array of SPLITFLAGs could be further compressed by taking into account impossible configurations, as done in [PH97].

4.4.2 Encoding Dependencies

Since an explicit encoding of the DAG describing the partial order could be too expensive, we encode it implicitly through a mechanism proposed by El-Sana and Varshney [ESV99]. This mechanism is based on a binary forest of vertices, and on a vertex enumeration scheme.

The forest of vertices is constructed in this way. The leaves of the forest correspond to the vertices of the reference mesh Σ_h, while the other nodes are in a one-to-one correspondence with the vertices created by the vertex-pair contractions. The two children of each internal node v_M correspond to the two vertices v'_M and v''_M which have been contracted to v_M. The roots correspond to the vertices of the base mesh Σ_0.

Vertices are numbered during the process of iterative vertex-pair contraction. The n vertices of the reference mesh Σ_h are labeled arbitrarily from 1 to n. The remaining vertices are labeled with consecutive numbers as they are created. In this way, if two modifications M and N are related in the partial order in such a way that $M \prec N$, then necessarily $v_M > v_N$. On the other hand, if $v_M > v_N$ then either N and M are not related, or $M \prec N$.

Given a mesh Σ_S corresponding to a closed subset S of modifications, the following rules are used to implement primitives *Insertion_Test*, *Removal_Test*, and *Dependency_Retrieval*:

1. A vertex-pair contraction M^- can be performed (i.e., no node N, with $M \prec N$, is in S) if and only if, for each vertex w adjacent to v'_M or v''_M, either w is a root, or the parent of w has a larger label than vertex v_M;

2. A vertex expansion M^+ can be performed (i.e., all nodes N, with $N \prec M$, are in S) if and only if, for each vertex w adjacent to v_M, w has a lower label than v_M;

3. If M^+ cannot be performed, then, for each vertex w adjacent to v_M which has a label greater than v_M, w must be expanded in order to make the expansion of v_M possible.

In order to prove the correctness of Rule 1, we have to show that, given a node $M \in S$:

1a. If some node N is in S, such that $M \prec N$, then some neighbor of v'_M or v''_M in Σ_S is an interior node of the forest and its parent has a label lower than v_M.
This is true because: we know that $v_N < v_M$ in the vertex enumeration; N^+ has been performed; and the two vertices resulting from expanding v_N are neighbors of v'_M or v''_M because N and M are related in the partial order. □

1b. If some neighbor of v'_M or v''_M in Σ_S is not a root and its parent v_N has a label lower than v_M, then node $M \prec N$ and $N \in S$.
This is true because: we know that N^+ has been performed, thus $N \in S$; and N is related with M in the partial order because they are spatially neighbors. Since $v_N < v_M$, then $M \prec N$. □

In order to prove the correctness of Rule 2, we have to show that, given a node $M \notin S$:

2a. If some node N, such that $N \prec M$, is not in S, then some neighbor of vertex v_M in Σ_S has a label greater than v_M.
This is true because we know that $v_N < v_M$ in the enumeration, and N^+ has not been performed. Thus, $v_N \in \Sigma_S$, and v_N must be a neighbor of v_M because M and N are related in the partial order. □

2b. If some neighbor v of vertex v_M in Σ_S has a label greater than v_M, then there exists a node N, such that N^+ expands v, $N \prec M$, and $N \notin S$.
This is true because vertices having a label greater than v_M are the result of contractions executed after M^- in the original sequence. Thus there is a node N such that N^+ expands v, and $N \notin S$ (since $v \equiv v_N \in \Sigma_S$); Since v is in the neighborhood of v_M, then N and M must be related in the partial order, therefore $N \prec M$. □

The correctness of Rule 3 follows immediately from Rule 2.

The space complexity of this encoding structure reduces to that of the binary forest, since the vertex enumeration can be encoded at no cost by simply storing vertices in an array according to their labels. The array entry corresponding to a vertex v_M stores: the index of the first child of v_M (by convention, the child having the largest label), and an index which points either to the parent of v_M, if v_M is a second child, or to the sibling of v_M, if v_M is a first child.

Note that the leaves of the forest do not need the first-child index. Therefore, our implementation consists of two separate arrays, one for the leaves and one for the internal nodes. Since the number of leaves is equal to the number n of vertices in the reference mesh Σ_h, the number of internal nodes is also bounded by n. The storage cost for the forest is equal to $3n$ indexes, i.e., $12n$ bytes. The overall space complexity for the data structure representing a non-manifold MT is thus equal to $20n + 12n = 32n$ bytes, plus the cost of encoding the base mesh Σ_0. However, this latter additional cost is generally negligible because the size of Σ_0 is $\ll n$.

Following El-Sana and Varshney [ESV99], the working data structure which represents the current mesh Σ_S is augmented by maintaining two more (small) integers for each vertex v, in order to speed-up the tests involved in primitives *Insertion_Test* and *removal_Test*. The two integers are the maximum vertex label of vertices adjacent to v in Σ_S, and the minimum

vertex label of parents of vertices adjacent v in Σ_S. Such additional information need not be maintained in the output data structure, thus it is stored in a separate array. In this way, the implementation of *Insertion_Test* and *Removal_Test* does not require accessing all adjacent vertices on the current mesh. Rule 1 reduces to testing whether or not the stored minimum labels on parents of incident triangles is larger than the label of v. Rule 2 reduces to testing whether or not the stored minimum label on v is greater than the label of v.

4.5 Implementation of mesh update operations

In this section, we explain how to implement the basic operations for selective refinement defined in Section 4.3.2 on a current mesh Σ_S encoded with the data structure described in Section 4.2.

Implementations of operations *Insertion_Test*, *Removal_Test* and *Dependencies_Retrieval* follow immediately from the mechanism described in Section 4.4.2. To this aim, the data structure encoding NMT dependencies provides parent-child relations between vertices, and the topological data structure encoding the current mesh provides vertex-adjacency relations.

Operations *Mesh_Refinement* and *Mesh_Coarsening* heavily rely on information encoded at nodes of the NMT, and need to be explained in more detail. In the following subsections, we sketch the main steps necessary to implement such two operations. More details on their implementation and related time complexity analysis are given in the Appendix.

4.5.1 Performing vertex expansions

Let Σ_S be the current mesh at a certain stage of the selective refinement algorithm, and let M^+ be a feasible vertex expansion in the current state. For the sake of simplicity, we denote $v = v_M$, $v' = v'_M$ and $v'' = v''_M$. We want to expand vertex v into vertices v' and v'' in Σ_S as specified in the codes associated with the corresponding modification M. The phases of the algorithm are described below.

- **Initialization.** Create the two new vertices v' and v'', retrieve relations $\mathrm{VT}(v)$ and $\mathrm{VV}^{trg}(v)$, and build some auxiliary data structures that will be used later.
- **Duplicate coincident triangles.** Process each triangle $t' = (v, w_1, w_2) \in \mathrm{VT}(v)$, such that $\textsc{splitflag}(t') = 2$, and transform it into two triangles incident in v' and in v'', respectively (see Figure 4.8a). Update relation TT at the edges of such triangles.

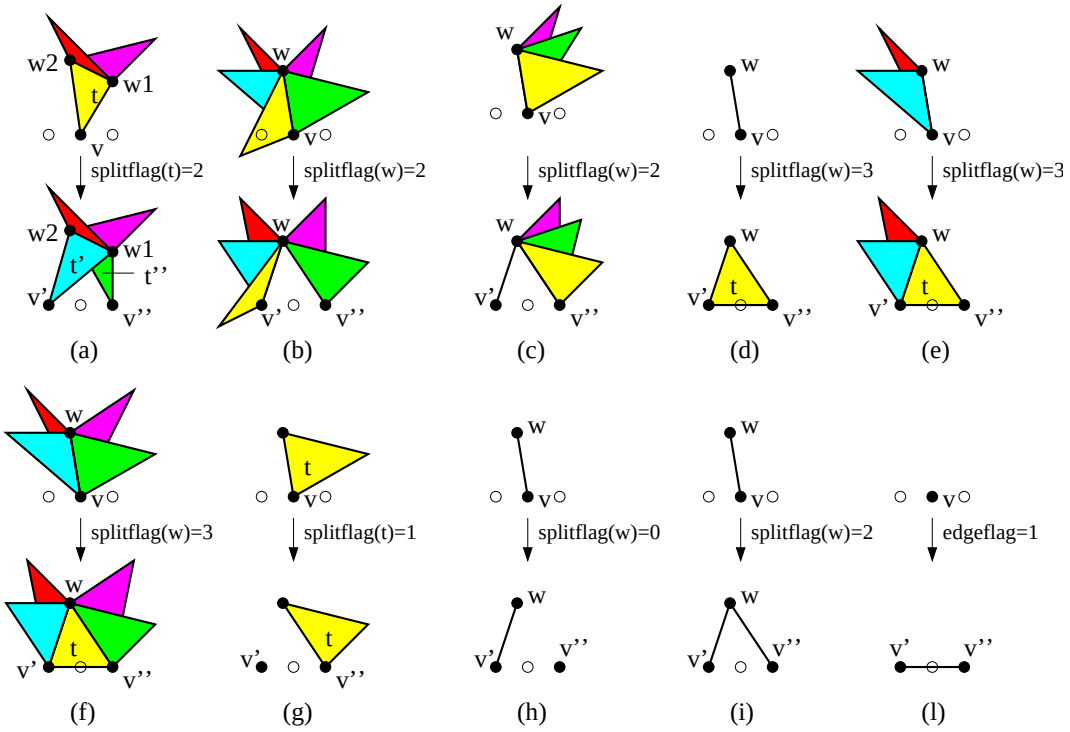


Figure 4.8: The possible configurations in vertex expansion.

After this phase, each pair of edges (v', w_i) and (v'', w_i) is still considered as one edge (w_i, v), having a unique cycle of triangles around it. Thus, the two new triangles temporarily coexist in the same cycle around (w_i, v), that will be later separated into two distinct cycles around edge (v', w_i) and (v'', w_i).

- **Separate coincident triangle-edges.** Process all triangle-edges (w, v) with $\textsc{splitflag}(w) = 2$ and transform each of them into two edges incident in v' and in v'', respectively (see Figure 4.8b,c). Moreover, update relation TT in order to separate the cycle of triangles around (w, v) into two cycles around (w, v') and (w, v''), respectively. If one of such cycles is empty, then update relation VV^{wir} in order to introduce a new wire-edge.

- **Expand wire-edges to triangles.** Process all vertices $w \in \mathrm{VV}^{wir}(v)$ with $\textsc{splitflag}(w) = 3$, and, for each of them, create a new triangle $t = (w, v', v'')$ (see Figure 4.8d).

- **Expand triangle-edges to triangles.** Process all vertices $w \in \mathrm{VV}^{trg}(v)$ with $\textsc{splitflag}(w) = 3$ and, for each of them, create a new triangle $t = (w, v', v'')$ (see Figure 4.8e,f). Moreover, update relation TT in order to separate the cycle of triangles around edge (w, v) into two cycles around (w, v') and (w, v''), respectively, and

insert t in both cycles.

- **Update triangles.** In any triangle $t \in \mathrm{VT}(v)$, replace v with either v' or v'', depending on SPLITFLAG(t) (see Figure 4.8g).
- **Update wire-edges.** Examine the vertices in $\mathrm{VV}^{wir}(v)$ and update relation VV^{wir} in such a way to replace each wire-edge (w, v) with either (w, v'), or (w, v''), or both, depending on SPLITFLAG(w) (see Figure 4.8h,i). We also set wire-edge (v', v'') if EDGEFLAG$= 1$ (see Figure 4.8l).
- **Adjust partial VT relation.** Check $\mathrm{VT}^*(v')$ and $\mathrm{VT}^*(v'')$, and eliminate any redundant triangle from them. This involves finding the edge-connected components of triangles incident in v' (and in v''), and then keep just one representative triangle for each component.

The overall time complexity of expanding vertex v is linear in the total number of triangles and wire edges incident at V, plus a term $O(A \log A)$, where A is the number of triangles incident at (v', v'') that are created by expansion. This last factor is due to the insertion in the proper position of each new triangle into the cycle of triangles incident at (v', v''), during phases "Expand wire-edges to triangle" and "Expand triangle-edges to triangle". However, A can be considered a constant in practical cases.

4.5.2 Performing vertex-pair contractions

Let M^- be a feasible vertex-pair contraction in the current state. We denote, as before, $v = v_M$, $v' = v'_M$ and $v'' = v''_M$. Also the algorithm to contract vertices v' and v'' to vertex v in Σ_S performs a number of phases, which are described in the following.

- **Initialization.** Create vertex v and build some auxiliary structures that will be used later.
- **Contract triangles to edges.** Delete each triangle t belonging to $\mathrm{VT}(v') \cap \mathrm{VT}(v'')$ from the mesh (see Figure 4.8e,f). If necessary, replace t with a wire-edge, by updating relation VV^{wir} (see Figure 4.8d). Triangles to be deleted are recognized because they contain both vertices v' and v''.
- **Glue triangle-edges.** Consider each pair of triangle-edges (w, v') and (w, v'') and merge the two cycles of triangles incident in them into one cycle around edge (w, v), by updating relation TT (see Figure 4.8b).

- **Glue triangle- and wire- edges.** Consider each pair of edges (w, v') and (w, v''), where one is a triangle- and the other one is a wire-edge, and delete the wire-edge by updating relation VV^{wir} (see Figure 4.8c).
- **Glue coincident triangles.** Consider each pair of triangles $t' = (w_1, w_2, v')$ and $t'' = (w_1, w_2, v'')$ and glue them into one triangle (w_1, w_2, v) (see Figure 4.8a). By convention, t' survives and is updated, while t'' is deleted. The triangles to be glued are found by checking the pairs of triangles that are mutually related through TT at an edge which is not incident v' or v''.
- **Update wire-edges.** Update relation VV^{wir} in order to transform wire edges incident in v' and v'' into wire-edges incident in v (see Figure 4.8h). This includes merging two wire-edges (w, v') and (w, v'') into one wire-edge (w, v) (see Figure 4.8i).
- **Update triangles.** Replace v' and v'' with v in the TV relation of all surviving triangles of $\mathrm{VT}(v') \cap \mathrm{VT}(v'')$ (see Figure 4.8g).
- **Adjust partial VT relation.** Eliminate redundant triangles from $\mathrm{VT}^*(v)$, by proceeding as explained in Section 4.5.1.

The overall time complexity of vertex-pair contraction is linear in the total number of triangles and wire edges incident at v' and v''.

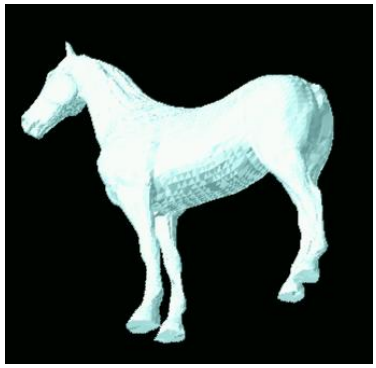
4.6 Results

Name	vertices	triangles	NMT nodes	triangles in root	wires in root
horse	11,135	33387	11034	4	5
stool	960	2746	869	68	31

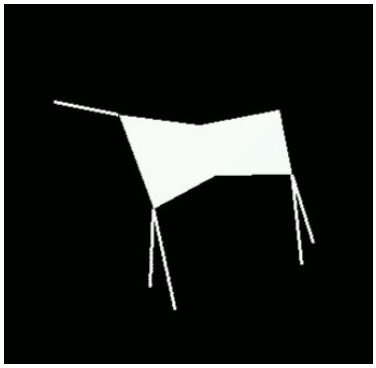
Table 4.2: Sizes of datasets and of corresponding NMTs.

In this section, we present some results of selective refinement, together with statistics about the behavior of the data structure and algorithms proposed in the previous sections.

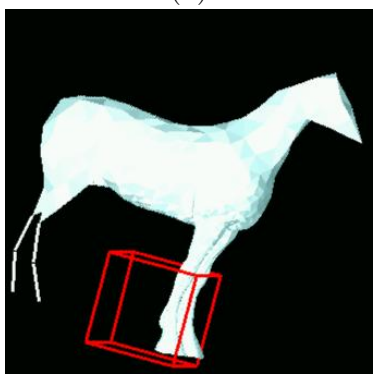
We have considered a number of input meshes representing either natural or mechanical objects. We have selected objects that contain some elongated or thin parts, possibly meeting at small joints, because such configurations are suitable to be represented through non-manifold, non-regular meshes at low level of detail. Here, we present two examples of datasets: one representing a horse, and another representing a stool. The input datasets are depicted in Figures 4.9a and 4.10a. Their size and the size of the NMT built on each of them are reported in Table 4.2.



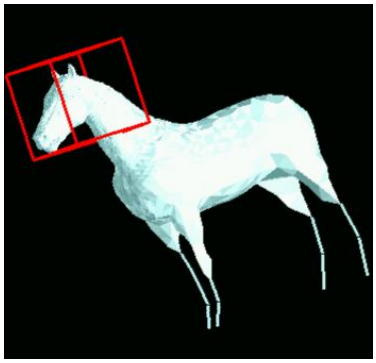
21618 triangles, 0 wire-edges
(a)



4 triangles, 5 wire-edges
(b)



3089 triangles, 4 wire-edges
(c)

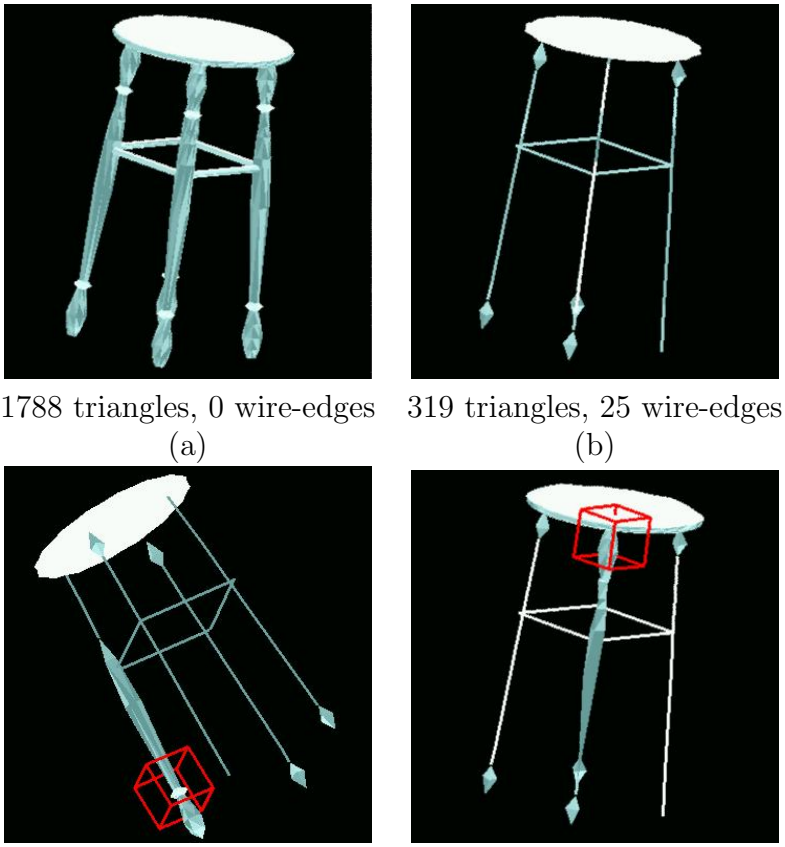


9494 triangles, 10 wire-edges
(d)

Figure 4.9: Some meshes meshes representing the horse at (a) the highest resolution, (b) the lowest resolution, (c) and (d) variable LOD.

An NMT is built through a simplification algorithm based on edge collapse, which iteratively simplifies a mesh by collapsing at each step the shortest edge. This is a straightforward approach to simplification, which produces an NMT with a moderate power of selectivity. Though this is certainly not an excellent choice of simplification strategy, it is sufficient to illustrate the behavior of our data structures. We wish to remark that more sophisticated non-manifold simplification criteria, which may highly enhance the power of selectivity, are beyond the scope of this work.

We run experiments that perform spatial selection queries, based on a box that moves through space containing the object. The Boolean function defining selective refinement requires that level of detail is maximum inside the box and arbitrarily low elsewhere. The box moves through space along a trajectory with a fixed step. At each step, selective refinement is applied starting at the mesh extracted at the previous position. We consider two trajectories: one is rectilinear and the other is random. Figures 4.9c, 4.9d, 4.10c,



1788 triangles, 0 wire-edges (a)

319 triangles, 25 wire-edges (b)

208 triangles, 27 wire-edges (c)

319 triangles, 25 wire-edges (d)

Figure 4.10: Some meshes meshes representing the stool at (a) the highest resolution, (b) the lowest resolution, (c) and (d) variable LOD.

and 4.10d show screenshots of the experiments for the two datasets. For each dataset, trajectory and step we counted the following quantities:

- NMT nodes: the number of nodes of the NMT used in the current set S;
- Triangles and wire edges in the extracted mesh;
- Contracted and expanded nodes: the number of nodes of the NMT traversed to obtain this set from the previous.

In Table 4.3 we report the average and standard deviation of such quantities computed on all frames for the two datasets and the two trajectories.

Experiment	NMT nodes		Triangles		Wire edges	
	avg	std.dev	avg	std.dev	avg	std.dev
Horse rectilinear	2,638.70	724.65	5,277.34	1,443.91	5.78	1.93
Horse random	2,332.50	130.39	4,670.33	261.60	5.98	0.95
Stool rectilinear	73.40	20.17	221.10	39.45	24.80	2.04
Stool random	76.40	31.30	233.40	67.39	26.80	2.82

(a)

Experiment	Contracted nodes		Expanded nodes	
	avg	std.dev	avg	std.dev
Horse rectilinear	86.90	67.30	140.24	123.75
Horse random	25.85	33.65	28.58	40.19
Stool rectilinear	8.30	11.75	10.50	14.00
Stool random	20.40	30.12	19.70	29.47

(b)

Table 4.3: Statistics on experiments (horse: 40 frames; stool: 10 frames). (a) Shows the number of NMT nodes in the closed set S corresponding to the current mesh Σ_S, and the number of triangles and wire-edges in Σ_S. (b) Shows the number of nodes removed from and inserted in S. For each item, the average and the standard deviation over all frames are reported.

From the statistics, it is evident that the number of nodes expanded or contracted during selective refinement is, on average, a small fraction of all used nodes. This happens because the incremental algorithm for selective refinement can take advantage of frame coherence. Also the number of triangles and wire edges traversed to update the current mesh is proportional to the size of the mesh itself.

Since the performance of algorithms described in subsections 4.5.1 and 4.5.2 depends on the manipulation of linked lists, we have also counted, for each collapse operation (hence for each node in the MT), the number of wire-edges and the number of fans of triangles incident at the vertex resulting from a collapse. Our experiments have shown that both such numbers (hence the length of linked lists) never exceed four. Moreover, the average number of fans of triangles incident at a vertex is less than 1.2; only 2% of vertices have incident wire edges, and the average number of edges incident at one such vertex is less than 1.25. Note that these statistics also indicate a small degree of "non-manifoldness" of meshes encoded in (and extracted from) an NMT, thus motivating our approach of working with structures that scale well to the manifold case.

Chapter 5

A Compact Data Structure for Volume MTs

In this Chapter we present the *Half-Edge Tree* (HET), a data structure for a tetrahedral multi-resolution model built through half-edge collapse, as proposed in [DFM+05b]. An HET consists of a collection of procedurally encoded half-edge collapses organized into a hierarchy of dependencies based on the vertices of the collapsed edges. We show that the HET is both a compact and efficient data structure. Since a half-edge collapse affects fewer tetrahedra than a full-edge collapse, variable-resolution representations with a finer granularity can be extracted from it.

The remainder of the Chapter is organized as follows. In Section 5.1 we briefly discuss half-edge and full-edge collapse operations. In Section 5.2, we describe the Half-Edge Tree, and discuss its correctness as a representation of the dependency relation in an LOD model and its storage cost. In Section 5.3, we describe basic algorithms for performing selective refinement queries on an HET. For comparison purpose, in Section 5.4 we review the Full-Edge Tree, an LOD model built through the full-edge operation. In Section 5.5, finally, we report results and experimental comparisons on several volume datasets.

5.1 Full-Edge and Half-Edge Collapse on Tetrahedral Meshes

In this Section we define the fundamental operations to simplify a tetrahedral mesh, with a special attention to the differences of the *influence region* of both.

A *full-edge collapse* consists of contracting an edge e with extreme vertices v' and v'' to

a new vertex v. The tetrahedra which are incident at v' or v'' become incident at v, and the tetrahedra incident at both v' and v'' are contracted into triangles. The set of the tetrahedra incident at v' or v'' form the *region of influence* of the edge-collapse (see Figure 5.1 from left to right). The reverse operation, called a *full-vertex split*, expands a vertex v into an edge e having its endpoints at v' and v''. A full-vertex split partitions the tetrahedra incident at v into two subsets, which are separated by a fan F of triangles incident at v. Tetrahedra of the two subsets are deformed to become incident at v' and v'', respectively. Triangles belonging to fan F become tetrahedra incident at both v' and v''. The set of tetrahedra incident at v form the *region of influence* of the full-vertex split (see Figure 5.1 from right to left).

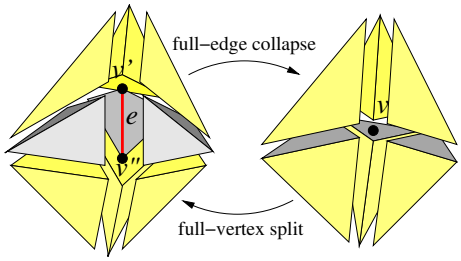


Figure 5.1: An example of a full-edge collapse and of a vertex split in a tetrahedral mesh. Edge $e = (v', v'')$ is collapses into the new vertex v

A *half-edge collapse* consists of contracting an edge $e = (v, w)$ to one of its extreme vertices, say w (see Figure 5.2 from left to right). The set of tetrahedra incident in v, but not in w, become incident in w, while the set of tetrahedra incident at both v and w become triangles incident in w. The set of tetrahedra incident at v form the *influence region* of the half-edge collapse. The reverse update with respect to a half-edge collapse is a *half-vertex split*, which expands a vertex w into an edge e by inserting the other extreme vertex v of e (see Figure 5.2 from right to left). Only a connected subset of the set of tetrahedra incident at w form the *influence region* of the vertex split. Such tetrahedra are deformed by replacing vertex w with v. The triangles in the fan of triangles incident at w which separate the influence region from the rest of the tetrahedra incident at w are expanded into tetrahedra incident at both v and w.

In our experiments, we have found that the influence region of a full-edge collapse contains about 32 tetrahedra on average, while that of a half-edge split about 17 tetrahedra on average. When such updates are used to build a multi-resolution model, larger influence regions generate more dependencies among updates, and this makes selective refinement propagate more slowly across the mesh. As a result, for the same LOD requirement, a mesh extracted through selective refinement from a multi-resolution model built through full-edge collapse will be larger than the mesh extracted from a multi-resolution model built through half-edge collapse. Moreover, a half-edge collapse does not generate new vertices,

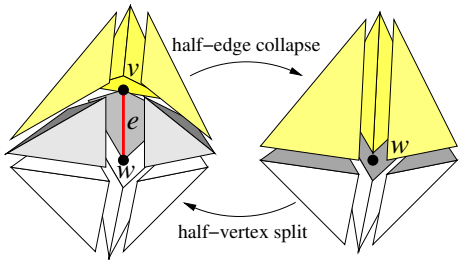


Figure 5.2: An example of a half-edge collapse and of a half-vertex split in a tetrahedral mesh. Edge e is collapsed to vertex w

and guarantees that the original field is interpolated exactly at all vertices of a simplified mesh. On the contrary, a full-edge collapse generates a new vertex and an interpolated field value each time it is applied. Full-edge collapse permits to optimize the position of the new vertex in order to minimize the error, e.g., through quadric error metrics [GH97], but this at the expenses of higher memory costs and of more sophisticated simplification algorithms.

5.2 The Half-Edge Tree

In this section, we describe a new data structure for encoding a tetrahedral LOD model generated through the iterative application of half-edge collapses to the mesh at full resolution. We call the resulting multi-resolution data structure a *Half-Edge Tree (HET)*. In the HET, we encode both the dependencies and the updates implicitly. We have developed a procedural encoding of an update, i.e., we encode how to perform half-edge collapse and its inverse, half-vertex split. The base mesh is stored separately in an indexed data structure with adjacencies. The HET has been generated in our experiments by using the simplification algorithm presented in [CCM+00].

5.2.1 Encoding Dependencies

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, that we have already described in Section 4.4.2, for the Non-Manifold Multi-Tessellation.

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

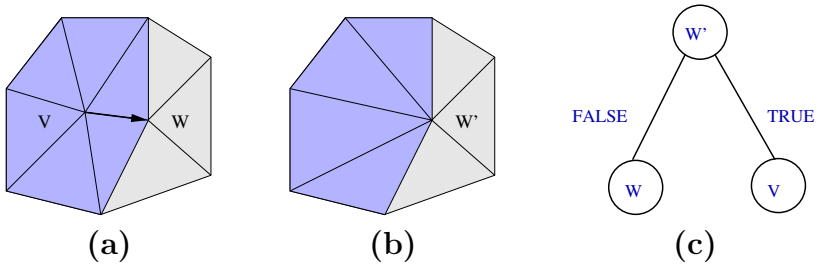


Figure 5.3: From (a) to (b): half-edge collapse. Note that w' is the same vertex as w but with a different label. From (b) to (a): half-vertex split operation. The polygon affected by updates is colored in blue. (c) Binary tree describing the half-edge vertex split updates. The left child is called the false-child of w' since it describes the same vertex as w'.

half-edge collapses and half-vertex splits applied on tetrahedral meshes.

We have modified the view 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 5.3).

The difference between half-edge collapse and full-edge collapse does not affect the vertex enumeration mechanism. Figure 5.4 shows a sequence of half-edge collapses and the corresponding vertex labels. Figure 5.5 shows the forest of view dependent trees corresponding to the half-edge collapses depicted in Figure 5.4.

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 5.6 (for false sons these links are shown by dashed lines).

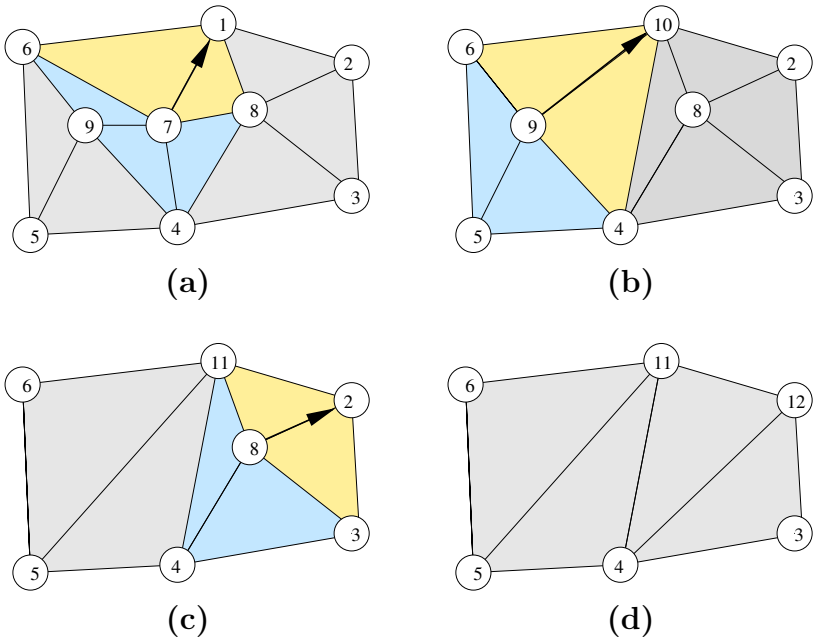


Figure 5.4: Sequence of half-edge collapses: (a) edge $(7, 1)$ is collapsed to vertex 1 that is labeled 10, (b) edge $(9, 10)$ is collapsed to vertex 10 that is labeled 11, (c) edge $(8, 2)$ is collapsed to vertex 2 that is labeled 12, (d) the base mesh. The triangles that removed as the result of the collapse operation are drawn in yellow, the triangles that are updated are shown in blue.

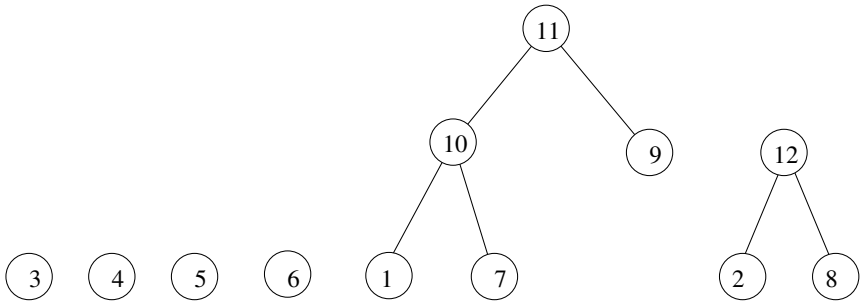


Figure 5.5: The binary forest correspond to half-edge collapse sequence depicted in Figure 5.4.

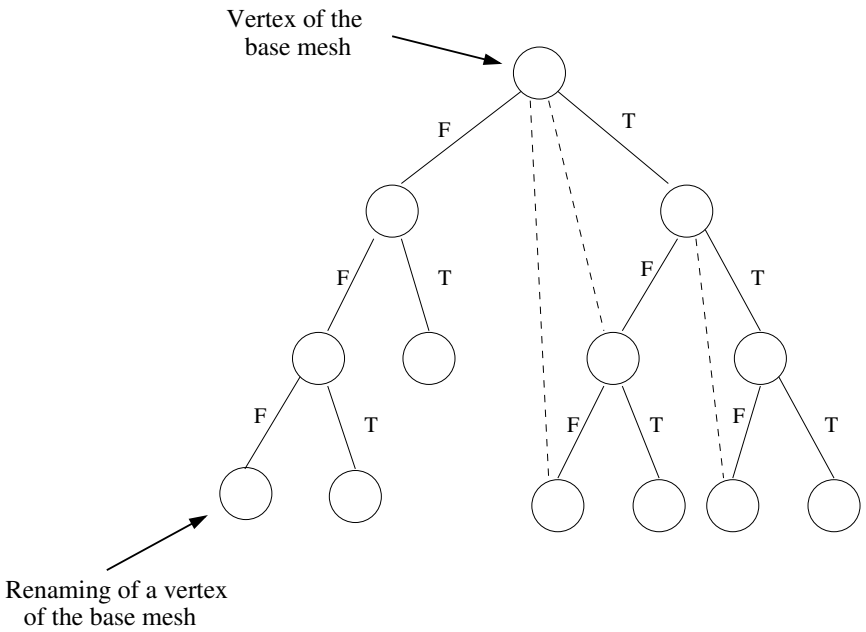


Figure 5.6: The right child of the node in binary tree is the true-child, while the left child is the false-child. Parent is a renamed copy of the left child. true parents of the left children (if they exist) are shown by dashed lines.

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 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.

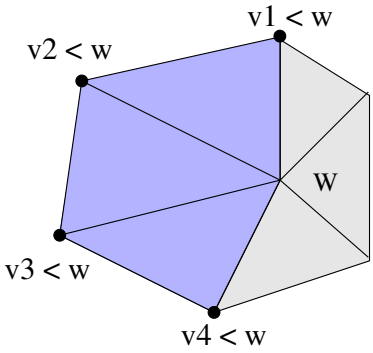


Figure 5.7: Labels of relevant neighbors of w should be less than label of w in order to perform split operation.

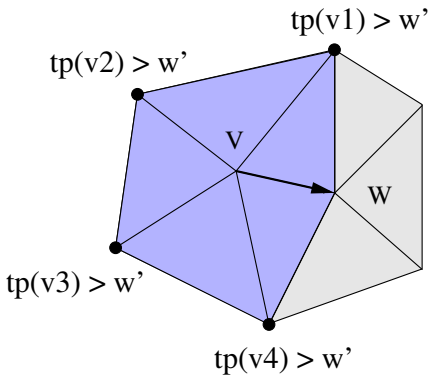


Figure 5.8: In order to collapse the edge (v, w) into vertex w', the labels of the true parents of neighbors of v should be larger than label of w' or no true parent. Note that $tp(v)$ denotes the true parent of vertex v.

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 5.8).

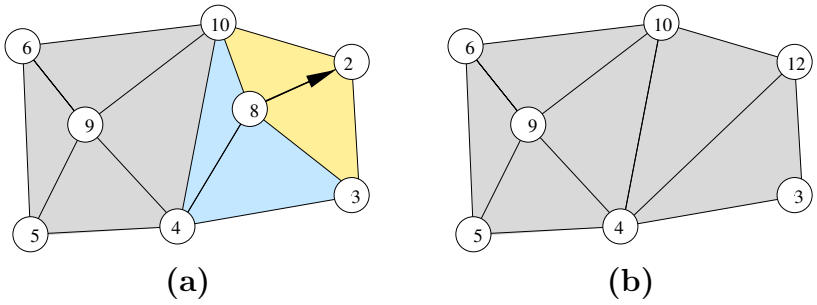


Figure 5.9: Applying legal collapse update of edge $(8, 2)$ on the mesh at step (b).

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.

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 img 5.4 and 5.5 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 5.4 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 5.9. 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 5.4 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 5.10. This would create two triangles which were not in the reference mesh.

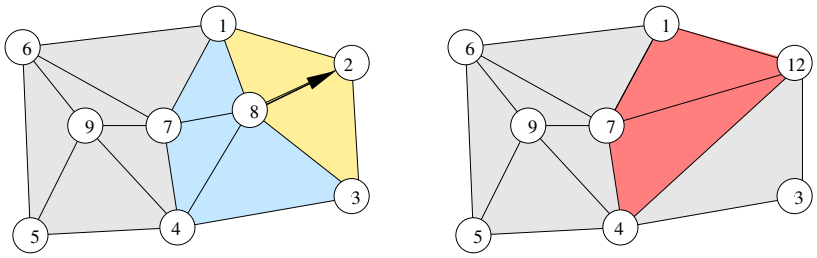


Figure 5.10: Applying illegal collapse update of edge $(8, 2)$ on the mesh of step (a). Illegal triangles are shown in red.

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;
- 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 $m \simeq n$).

5.2.2 Encoding Updates

Each update is described implicitly by storing information sufficient to perform a half-edge collapse and its inverse half-vertex split on any mesh extracted from an HET. We consider an update u corresponding to a half-edge collapse that contracts edge $e = (v, w)$ into vertex $\hat{w}$, and to its inverse split of vertex $\hat{w}$ into edge $e = (v, w)$. Note that the structure of the HET is sufficient to retrieve information necessary to perform a half-edge collapse. since the two endpoints v and w of edge e can be simply retrieved through the labels of the children of u.

Most of the information stored at a node are needed for performing a half-vertex split. In this case, we need to find the vertex to be split, the position and field value of the new vertex generated by the split, and the influence region of the split, i.e., the tetrahedra in Σ_1, where $u = (\Sigma_1, \Sigma_2)$. The vertex to be split is found through the label of u, but the remaining information must be stored. In particular, the mechanism for finding Σ_1 is based on a traversal of the current mesh through adjacencies, starting at a given face f. Thus, the following information are encoded for each update u in the HET:

- *Geometric information*: an offset vector, which is used to find the coordinates of vertex v from those of vertex w.
- *Field information*: an offset value, which is used to find the field value of vertex v from that of vertex w.
- *Error information*: the approximation error introduced by performing the edge collapse, which is used to decide whether or not to perform the collapse/split represented by u. The error value associated with an update u is computed as the maximum of the errors associated with the tetrahedra forming Σ_1. Here, we store both the *field error*, which measures the accuracy with which a scalar field approximation provided by a simplified domain decomposition approximates the original field data, and the *iso-surface error*, which measures the accuracy with which the iso- surfaces extracted from a simplified representation of the field approximate the iso-surfaces extracted by the original (full-resolution) representation (see [CCM+00, CDFM+04] for a thorough analysis of such approximation errors).
- *Connectivity information*, which encode the traversal of the region of influence Σ_1 of the vertex split:
  - the index of a tetrahedron t within Σ_1 containing a starting face f;
  - the index of starting face f within t;
  - a bit stream describing a traversal of Σ_1 starting at f.

The bit stream contains two bits for each tetrahedron in Σ_1, and is built as follows. We start from the tetrahedron t in Σ_1, containing face f, and we traverse the adjacency graph

G_u corresponding to the region of influence Σ_1, in which nodes correspond to the tetrahedra of the region of influence Σ_1 and the arcs to the faces of such tetrahedra. We label the faces of the tetrahedra encountered in the traversal in breadth-first order. A face of a tetrahedron, which is internal to Σ_1 is labeled 1, while it is labeled 0 otherwise. Note that, for a given tetrahedron t, we need to label only two faces, since one face of t is opposite to w and, thus, it is for sure on the boundary of Σ_1, and another face is the face from which we entered t during the traversal.

Note that a half-vertex split generates a set of new tetrahedra that have to be labeled. In order to avoid storing labels for new tetrahedra, we assign to them the label of the new vertex v, plus few additional bits. The maximum number of new tetrahedra inserted by a split update is 32, therefore we need 5 additional bits to build their labels.

5.2.3 Storage Cost

In this subsection, we analyze the storage cost of the HET encoding data structure. We disregard the cost of the base mesh, since it is negligible with respect to the size of the mesh at full resolution. The cost of the HET encoding structure corresponds to that of maintaining the forest of updates. We report separately the cost for encoding the structure of the forest, and that for encoding each update. We use 2 bytes for (quantized) real numbers and 4 bytes for indexes/pointers.

For a reference mesh with n vertices, the forest has n_{in} internal nodes and n leaf nodes, with $n_{in} \simeq n$. Thus the storage cost for the forest is equal to $8\ n_{in} + 4\ n \simeq 12n$ bytes.

Update information need to be stored only at internal nodes. The cost of encoding the coordinates and the field value is equal to 8 bytes, while the error information (field and iso-surface errors) is compressed on 4 bytes, as for the FET [CDFM$^+$04]. An index of the first tetrahedron t within Σ_1 and an index of starting face f within t require a total of 5 bytes. If u^- contains k tetrahedra, then the bit stream contains $2k$ bits. Our experiments have shown that we can safely assume $k = 15$, thus, 30 bits for the stream, i.e., 4 bytes. Summing up all the contributions, the storage cost for the information associated with a single update turns out to be equal to 21 bytes. Therefore, the total cost of the HET is equal to $12n + 21n = 33n$ bytes.

5.3 Selective Refinement on an HET

As we have seen, the basic operation on an LOD model, the selective refinement, consists of extracting a mesh satisfying some application-dependent requirements based on level of detail, such as approximating a scalar field with a certain accuracy which can be uniform

or variable in space.

In terms of the HET, adding an update u to U' and modifying Σ accordingly consists of performing a vertex split on the extracted mesh Σ, while removing an update u from U' and modifying Σ accordingly consists of applying an half-edge collapse to Σ. A half-vertex split of a vertex $\hat{w}$ in the current mesh Σ into edge $e = (v, w)$ is *feasible* if each relevant neighbor of $\hat{w}$ has a label lower than $\hat{w}$. This is equivalent to test if all direct ancestors of the corresponding update are in U'. Conversely, a half-edge collapse of edge $e = (v, w)$ into vertex $\hat{w}$ is feasible if all neighbors of vertex v (except w) have either a true parent with a label greater than $\hat{w}$, or no true parent.

We report below the primitives that are sufficient to implement an incremental selective refinement algorithm on an HET. Note that a subset of such primitives is sufficient to implement the top-down algorithm, since no coarsening of the currently extracted mesh is performed. The selective refinement primitives are defined as follows:

- Primitive `Collapse` performs a feasible half-edge collapse (corresponding to an update u) on the current mesh.
- Primitive `Split` performs a feasible half-vertex split (corresponding to an update u) on the current mesh.
- Primitive `CollapseTest` checks the feasibility of a half-edge collapse on the current mesh.
- Primitive `SplitTest` checks the feasibility of a half-vertex split on the current mesh; if the split is not feasible, then it returns *one* of the vertices which must be split in order to achieve feasibility.
- Primitive `ForceSplit` performs a half-vertex split on the current mesh, after having recursively performed all the splits needed to achieve its feasibility.

5.3.1 Implementation of selective refinement on an HET

In this subsection, we describe the implementation of the selective refinement primitives introduce before on the HET. We assume that update $u = (\Sigma_1, \Sigma_2)$ corresponds to split vertex $\hat{w}$ into edge $e = (v, w)$, when regarded as a refinement update, and to the corresponding half-edge collapse, when regarded as a coarsening update. We assume that a Half-Edge Tree T is provided with the following access functions, that are easy to implement on the HET data structure:

- Function `RetrieveCollapseInfo`, given an update u, retrieves the vertices v, w of the collapsed edge and the index of the renamed vertex $\hat{w}$.

- Function `GetSplitInfo`, given an update u, retrieves the vertex w to be split and the other vertex v of the edge to be created, the initial tetrahedron t, the starting face f, and the bit stream.

We also assume to encode the current mesh Σ into the extended indexed data structure with adjacencies, and that this data structure is provided with standard access functions to retrieve topological relations (e.g., the tetrahedra incident at a vertex, the vertices adjacent to a vertex), as well as standard update functions to insert/delete tetrahedra and vertices.

Primitive `Collapse`. To perform a half-edge collapse by contracting edge (v, w) to vertex $\hat{w}$, we retrieve all the tetrahedra incident at v from the current mesh, and replace vertex v within these tetrahedra by vertex $\hat{w}$. The tetrahedra collapsed into triangles are removed from the current mesh and adjacencies are updated accordingly. A pseudo-code description of primitive `Collapse` is shown in Figure 5.11. Function `ReplaceVertex` modifies a tetrahedron in the current mesh by replacing one of its vertices, while function `RenameVertex` renames an existing vertex.

```
Collapse(T : HET, u : update, Σ : mesh) {
   (v, w, ŵ) = RetrieveCollapseInfo(T, u);
   L = IncidentTetra(Σ, v);
   for each tetrahedron t in L do
      if IsIncident(t, w) then
         DeleteTetrahedron(Σ, t);
      else ReplaceVertex(Σ, t, v, ŵ);
   RenameVertex(Σ, w, ŵ);
   DeleteVertex(Σ, v);
}
```

Figure 5.11: Pseudo-code description of primitive `Collapse`.

Primitive `Split`. In order to split vertex $\hat{w}$ into an edge $e = (v, w)$, we start from the starting tetrahedron t encoded and from its boundary face f, and use the bit stream to retrieve all tetrahedra of Σ_1 by visiting them in the same order as they have been visited when creating the bit stream. The tetrahedra of Σ_2 found in this way, are updated by replacing vertex $\hat{w}$ with vertex v. New tetrahedra, incident in e, are inserted and adjacencies are updated accordingly. The pseudo-code description of primitive `Split` is shown in Figure 5.12. The breadth-first traversal of Σ_1 is performed by using a queue Q, where each element of queue Q contains a tetrahedron and the face used to enter it. Index b is used to scan the bit stream: we advance one position for each element extracted from the queue. Auxiliary function are defined as follow:

- Function `BuildNewTetrahedron` constructs a tetrahedron from a given face and a given vertex, not belonging to the face.

- Function `NextTetra` returns the other tetrahedron sharing face f with t. Index i is a progressive number used to generate the labels for the new tetrahedra by starting adding a few bits to the label of v.

```
Split(T : HET, u : update, Σ : mesh) {
   (v, w, t, f, bitStream) = GetSplitInfo(T, u);
   i = 0;
   InsertVertex(Σ, v);
   if IsVertex(w, f) then
      t'' = BuildNewTetrahedron(f, v);
      Label(t'', (v, i));
      i = i+1;
      InsertTetrahedron(Σ, t'');
   CreateEmptyQueue(Q);
   Enqueue(Q, (t, f));
   b = 0;
   while not IsEmptyQueue(Q) do
      (t, f) = Dequeue(Q);
      ReplaceVertex (t, w, v);
      for each face f' of t do
         if bitStream[b] == 1 then
            t' = NextTetra(Σ, f, t);
            Enqueue(Q, (t', f'));
         else if IsVertex(w, f) then
            t'' = BuildNewTetrahedron(f, v);
            Label(t'', (v, i));
            i = i+1;
            InsertTetrahedron(Σ, t'');
         b = b+1;
}
```

Figure 5.12: Pseudo-code description of primitive `Split`.

Primitive `CollapseTest`. This primitive simply applies the rule for half-edge collapse described in Section 5.2.1. Figure 5.13 shows the pseudo-code description of primitive `CollapseTest`. Function `TrueParent` returns the true parent of a vertex in the HET.

Primitive `SplitTest`. This primitive applies the rule for half-vertex split described in Section 5.2.1. In the first step, the relevant neighbors of vertex $\hat{w}$ to be split are found. The idea is to find the tetrahedra of Σ_2 in the current mesh: the relevant neighbors of $\hat{w}$ are the vertices of such tetrahedra different from $\hat{w}$. If the split of $\hat{w}$ is not feasible, then such traversal will fail: in this case, we will find, and return, a vertex that must be split before $\hat{w}$. In particular, if the starting tetrahedron t is not present, then we fail and return the vertex whose splitting creates t. Function `Creator` finds this vertex by starting

```
CollapseTest(T : HET, u : update, Σ : mesh) {
   (v, w, ŵ) = RetrieveCollapseInfo(T, u);
   L = AdjacentVertices(Σ, v)
   for each vertex q in L
      if q ≠ w and not IsRoot(T, q) and
         ŵ > TrueParent(T, q)
            return (false);
   return (true);
}
```

Figure 5.13: Pseudo-code description of primitive `CollapseTest`.

from the label of t. If, following the directions contained in the bit stream, we reach a tetrahedron that has some vertex z with a label greater than $\hat{w}$, then we fail, and return z. If the traversal arrives is completed, then the vertex split is feasible. Figure 5.14 shows the pseudo-code description of primitive `SplitTest`.

```
SplitTest(T : HET, u : update, Σ : mesh) {
   (v, w, t, f, bitStream) = GetSplitInfo(T, u);
   if t ∉ Σ then
      return Creator(t);
   CreateEmptyQueue(Q);
   Enqueue(Q, (t, f));
   b = 0;
   while not IsEmptyQueue(Q) do
      (t, f) = Dequeue(Q);
      for each vertex q of t do
         if q ≠ w and q > w then
            return q;
      for each face f' of t do
         if bitStream[b] == 1 then
            t' = NextTetra(Σ, f, t);
            Enqueue(Q, (t', f'));
         b = b+1;
   return (null); }
```

Figure 5.14: Pseudo-code description of primitive `SplitTest`.

Primitive `ForceSplit`. In order to force the split of a vertex $\hat{w}$, we first check the feasibility of splitting $\hat{w}$ through primitive *SplitTest*. As long as some update u' is returned, then we recursively force splitting u'. Finally, we split $\hat{w}$ by using primitive `Split`. Figure 5.15 shows the pseudo-code description of primitive `ForceSplit`.

```
ForceSplit(T : HET, u : update, Σ : mesh) {
   u′ = SplitTest(T, u, Σ);
   while u′ ≠ null do
      ForceSplit(T, u′, Σ);
      u′ = SplitTest(T, u, Σ);
   Split(T, u);
}
```

Figure 5.15: Pseudo-code description of primitive `ForceSplit`.

5.4 The Full-Edge Tree

In this Section we shortly describe the Full-Edge Tree as proposed in [CDFM+04]. As usual, we have to take care of encoding the modifications and the dependency relation among them.

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 4.4.2

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 4.4.2). 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.

5.5 Experimental Results and Comparisons

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

- *Buckminster Fullerene (Buckyball)* (Robert Haimes, MIT) represents a carbon molecule having 60 atoms arranged in the form of a soccerball, 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

Table 5.1: Space required for storing each dataset with different data structures.

Data Set	Vertices (K)	Tetra (K)	HET (Kb)	Index (Kb)	Ratio	Ind-Adj (Kb)	Ratio	FET (Kb)	Ratio
Buckyball	256	1221	7979	21583	2.70	42142	5.28	7773	0.97
Plasma	268	1280	8175	22626	2.77	44178	5.40	8046	0.98
Bluntfin	40	217	1364	3795	2.78	7430	5.45	1342	0.98
Post	106	602	3428	10472	3.06	20521	5.99	3311	0.97
Fighter	14	68	580	1204	2.08	2354	4.06	570	0.98
San Fernando	370	2019	11406	35267	3.09	69055	6.05	11103	0.97

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.

Table 5.1 reports the characteristics of the above data sets, together with their storage cost by using the following data structures: the Half-Edge Tree (HET), the indexed data structure (Index), the extended indexed data structure with adjacencies (Ind-Adj), and the full-edge tree (FET) proposed in [CDFM+04]. Columns marked *Ratio* report the ratio between the storage cost for a given data structure and the corresponding storage cost with our HET. Note that original unstructured data sets are described as an indexed data structure, while meshes extracted both from our HET and from the FET are encoded by the extended indexed data structure with adjacencies. Unlike other data structures, the HET has variable size and thus the ratio is not always the same. Our data structure achieves a compression factor between 52% and 68% with respect to the indexed data structure and a compression factor between 73% and 83% with respect to the extended indexed data structure with adjacencies.

We have compared the performances of the HET and of the FET by considering the following selective refinement queries:

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 a 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 5.16, 5.17 and 5.18 reports 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 two regular data sets (BuckyBall and Plasma), from 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. Figures 5.16 report the number of tetrahedra in the meshes extracted at uniform LOD as a function of the error threshold. Figures 5.17 report the number of tetrahedra in the meshes extracted at a variable LOD based on the field value as a function of the isovalue. Figures 5.18 report the number of tetrahedra in the meshes extracted at a variable LOD based on a box as a function of the error threshold. For each experiment, we report the results obtained from both the HET and the FET, as well as the ratio between the size of the results obtained from the HET and the FET (ratio is an average between the two data sets considered in each figure). On average, we see that the HET extracts meshes that, for a given error threshold, are 34% smaller than those extracted from the FET, and this gain can be up to about 75% for large error thresholds (coarse approximation).

Figures 5.19 (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 5.20 (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 5.21 (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.

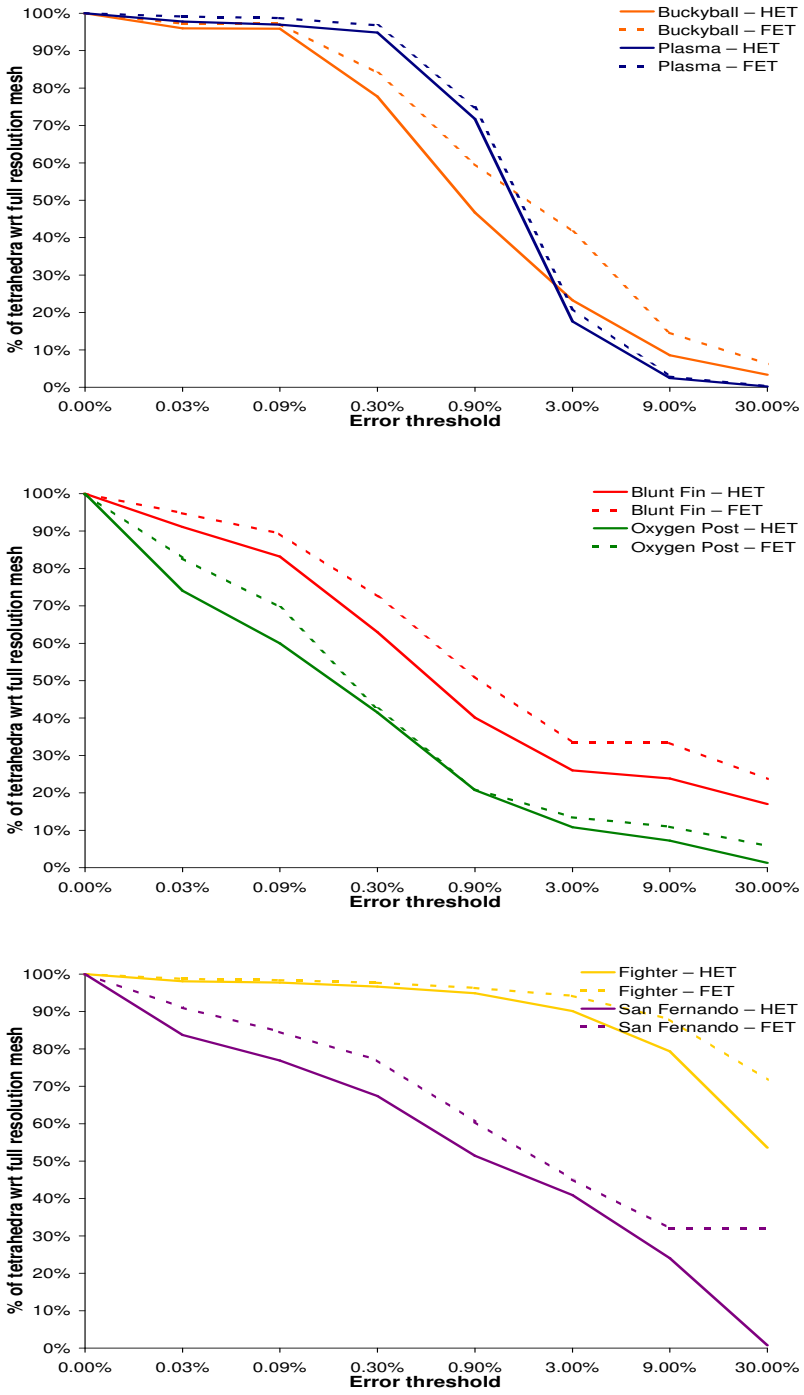


Figure 5.16: Uniform LOD query: number of tetrahedra in the extracted mesh as a function of the error threshold. From top to bottom: regular, curvilinear and irregular data sets.

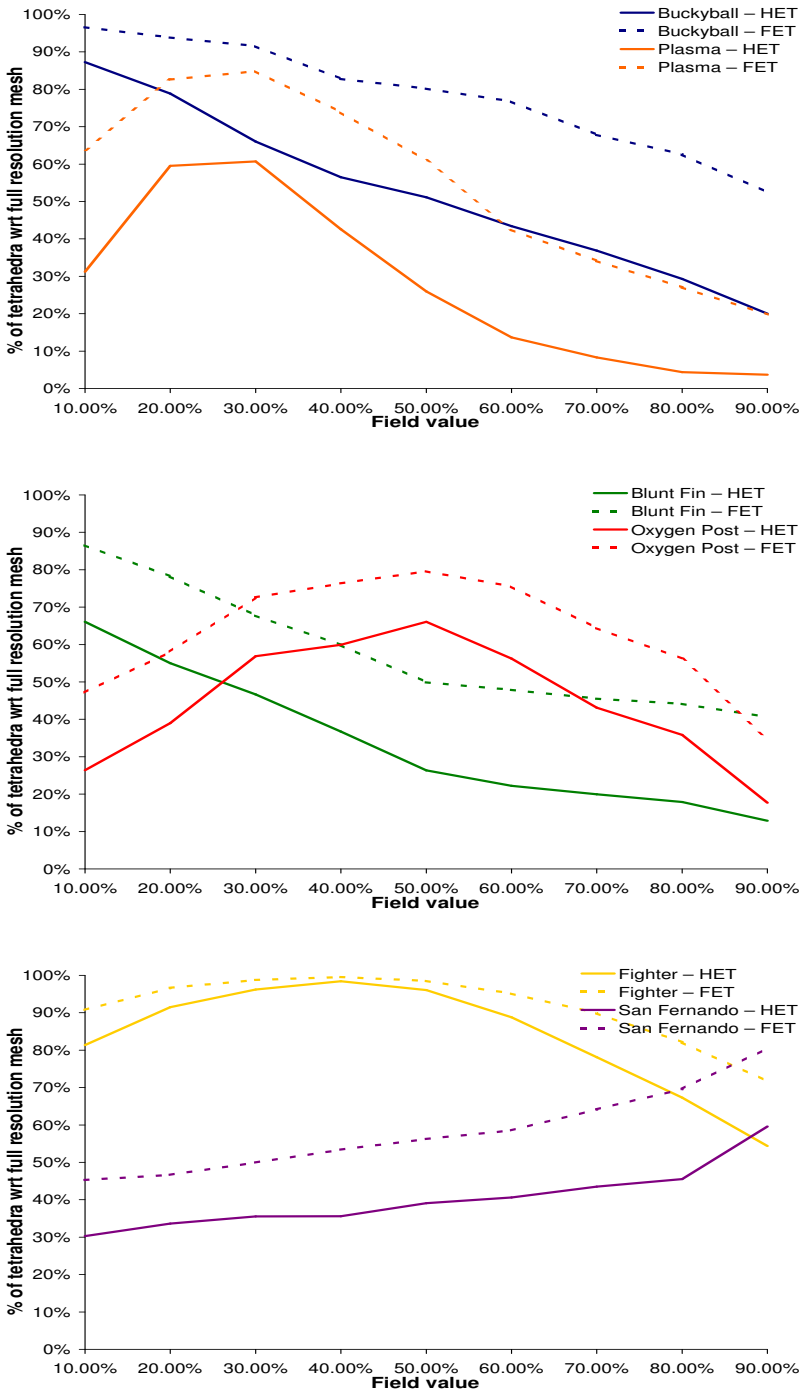


Figure 5.17: Variable LOD query based on the field value: number of tetrahedra in the extracted mesh as a function of the isovalue. From top to bottom: regular, curvilinear and irregular data sets.

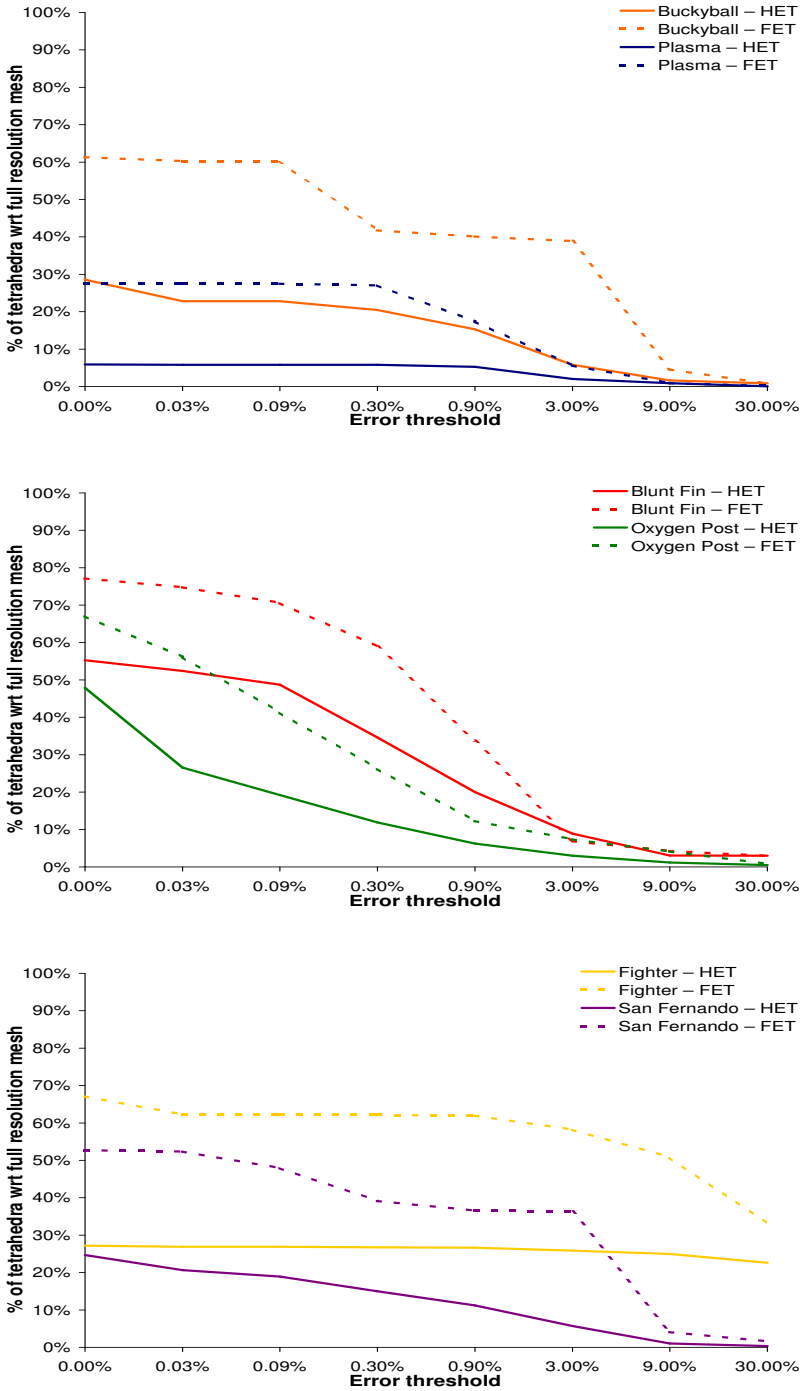
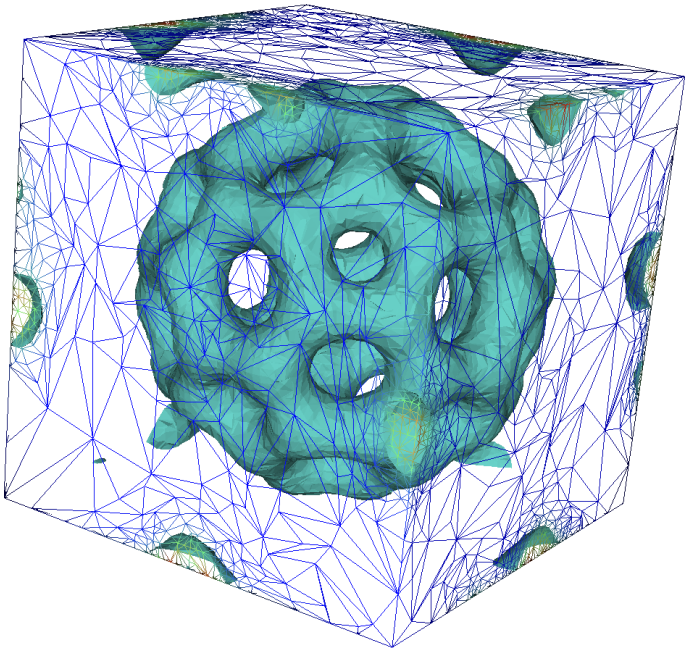
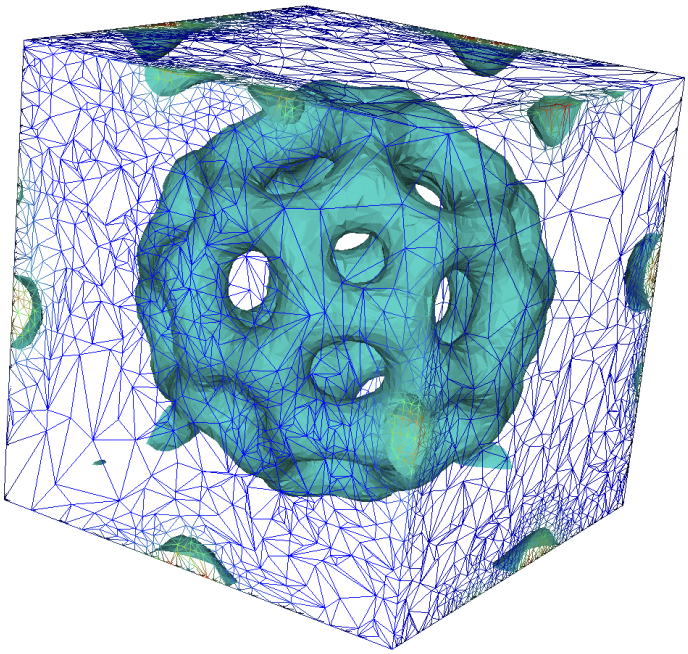


Figure 5.18: Variable LOD query based on a box as focus set: number of tetrahedra in the extracted mesh as a function of the error threshold. From top to bottom: regular, curvilinear and irregular data sets.

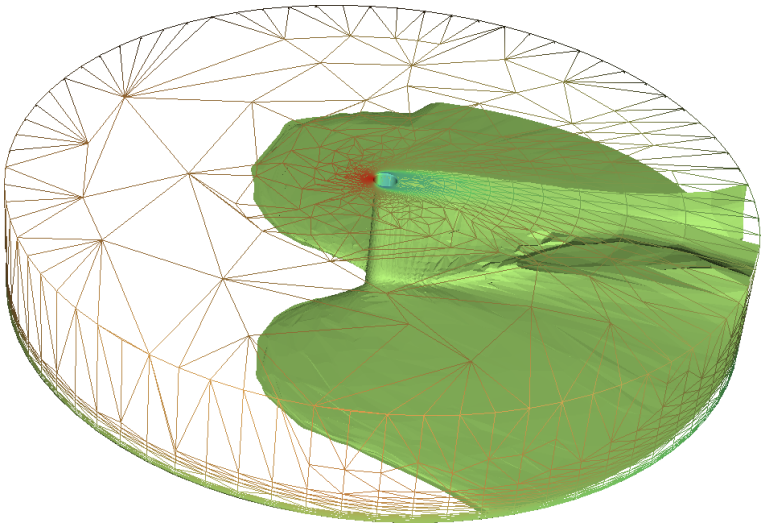


(a)

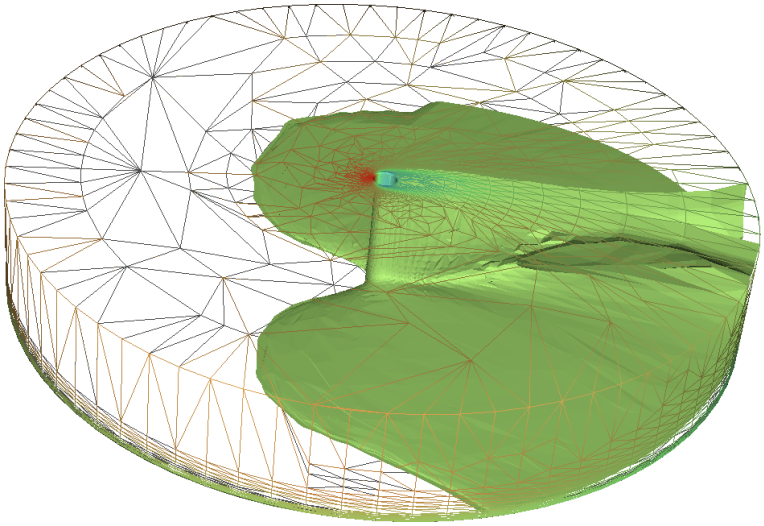


(b)

Figure 5.19: 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, (b) Full-Edge MT

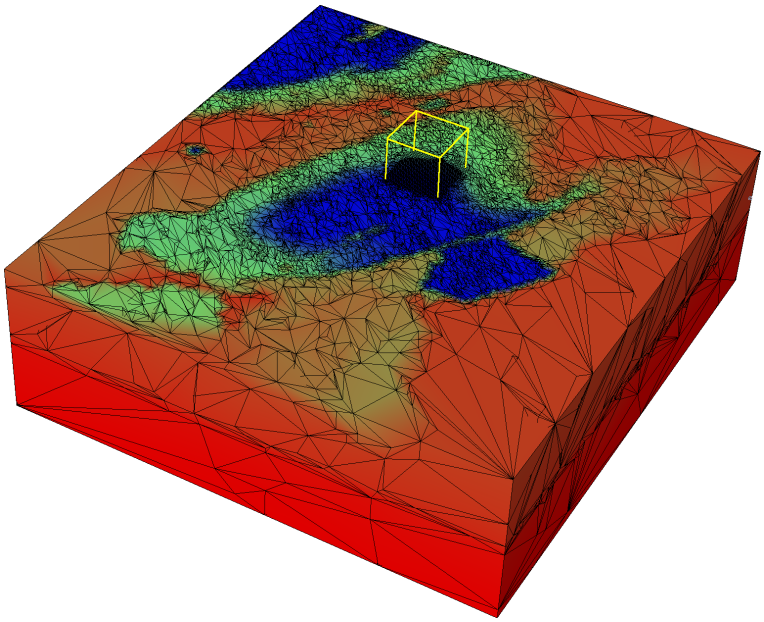


(a)

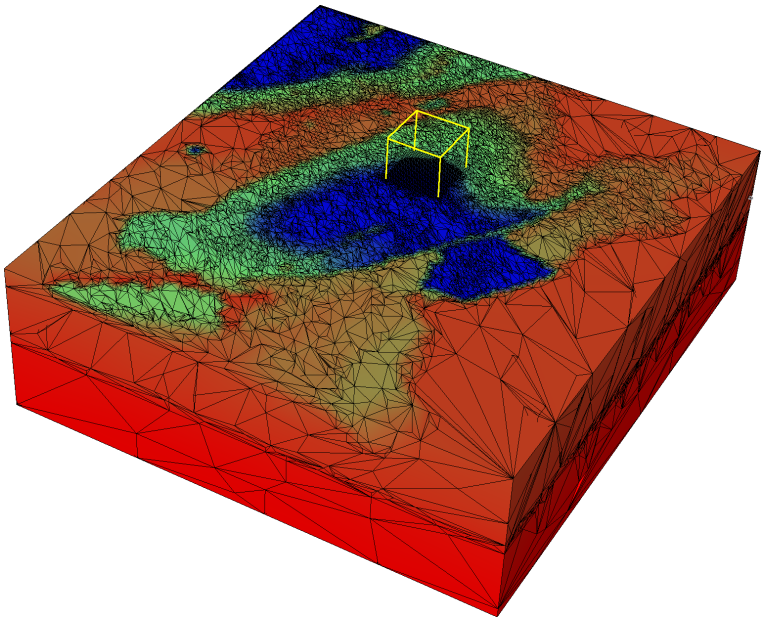


(b)

Figure 5.20: 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, (b) Full-Edge MT



(a)



(b)

Figure 5.21: 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, (b) Full-Edge MT

Chapter 6

A Client/Server Approach to Multi-resolution Modeling

In this Chapter we present a new general framework for a client/server application based on compact data structure for two- and three-dimensional MTs. We focus on the class of compact representation based on the *View Dependent Tree* (VDT) [ESV99], such as the NMT described in Chapter 4 and the FET [CDFM+04], or extended versions of it, as the HET, described in Chapter 5. This general framework is based on the ideas sketched in [DFM+06].

The Chapter is organized as follows. In Section 6.1, we recall the basic ingredients common to multi-resolutions data structure based on the view dependent tree. In Section 6.2, we describe the general framework for a client/server system which allows querying a multi-resolution model from remote. We discuss general assumptions we have done (Subsection 6.2.1) and we detail the system. Section 6.3 describes how the selective refinement primitives need to be modified in a network environment.

6.1 Data Structures for Compact MTs

In this Section we briefly recall the main characteristics of compact MTs based on the view dependent tree; for more details, see Chapter 4 for the Non-Manifold MT and Chapter 5 for the Half-Edge Tree. In what follows, we refer to this class of compact data structures as *View-Dependent MT* (VDMT).

A VDMT has four major ingredients:

- a *base mesh* at low resolution; this is usually the result of a simplification process,

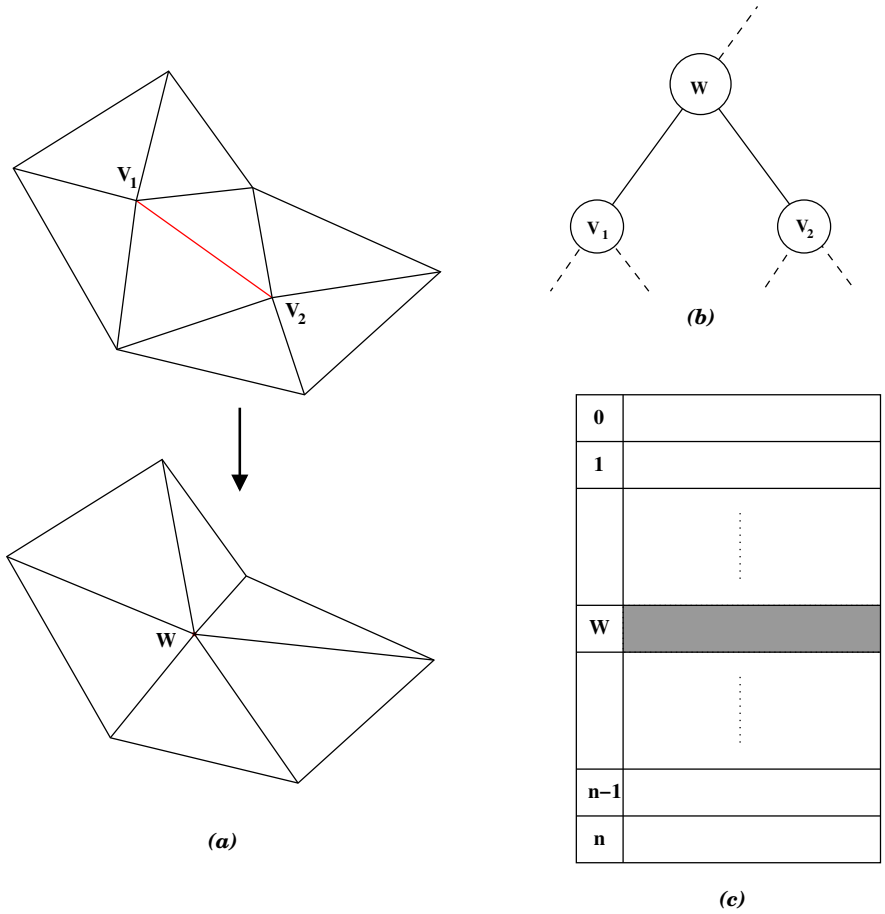


Figure 6.1: Use of labeling in VDMT. (a) A full-edge collapse. (b) The corresponding part of the view-dependent tree. (c) The set of updates.

and its size depends on the parameters and the roughness of simplification;

- an *enumeration scheme*, which assigns to every vertex in the VDMT a unique identifier; the enumeration is usually done during the simplification phase, by starting from the vertices of the mesh at full resolution and labeling every new vertex inserted in the mesh by the simplification algorithm;

- a *forest of binary trees*, which encodes the partial dependency relation among the different modifications; the nodes of the trees have the same labels given from the enumeration scheme;

- a *set of updates*, encoded in a compact way according with the different context. This set is indexed according to the labels of the enumeration scheme, and for every label of an internal node in the forest, there is a corresponding update stored.

In Figure 6.1 we show an example of the use of the labeling process in a VDMT: the collapse of the edge $[v_1, v_2]$ to the vertex w generates a part of the view dependent tree with the same labels, and, in the set of updates, the corresponding update is stored with label w. Notice that both v_1 and v_2 are less than w, according to the enumeration scheme.

6.2 A General Framework for Client-Server MTs

In this Section we explain the basic ideas of the client/server framework we design for the View Dependent MT. We start from some general assumptions about the process (Subsection 6.2.1) and we describe in the remaining sections the data structures on the server side, the ones on the client side and the scheme of the communication session.

The basic idea is the following. At the beginning, the whole VDMT is stored only on the server, while just the base mesh and some basic structural information are transferred to the client. The client starts sending selective refinement queries to the server for a mesh at a certain level of detail. The server does not send the mesh to the client. Instead, it sends the updates operations needed to modify the current representation existing on the client (initially, the base mesh), and transforming it into the mesh at the desired level of detail. After some queries, a part of the VDMT has been reconstructed on the client side. In this situation, when the client needs a mesh at a given level of detail, first it tries to obtain such mesh from the part of VDMT that is already stored in its local memory. If this is not possible, the client sends a request to the server. The server sends to the client only the missing updates.

6.2.1 General Assumptions

Since we want to define a system that could be easily generalized to a WEB environment, we need to assume that the server can satisfy the requests of a number of clients, without having to store all the time the information about the state of each client. The server needs only to maintain the VDMT in an appropriate format such that, in case of a request from an authorized client, it can quickly send the appropriate information without checking the correctness of the information according to the client state.

To fulfill this condition, the client side cannot be considered as a simple "visualization machine", which directly displays the data coming from the server. We assume that the client has sufficient storing and computing capacities to progressive rebuild on its side the parts of the multi-resolution structure that are necessary to perform selective refinement queries. The most important of these two characteristics is the computing power, which has impact on the speed of selective refinement queries. As for as the storing capabilities

are concerned, we can adapt the caching mechanism to the available space. In case the client has a limited space, this is also an important factor that can affect the visualization process, because the client needs to wait all the time for information sent from the server side.

In this proposal, we do not consider of the soundness of the transmission session as an issue. We assume that the communication channels are robust, like TCP/IP connections. Also we do not deal with security: what are the conditions or the credentials that a client must have to gain the rights to access to a VDMT, eventually downloading the entire multi-resolution model? These issues are out of the scope of this thesis.

6.2.2 Clustering the View-Dependent tree

As we have seen before, the forest of binary trees of the view-dependent tree, used to encode the partial dependency relation among the updates, is a data structure that contains a number of labels in the order of the number n of vertices of the mesh at full resolution. Assuming that the number of internal nodes n_{int} of the trees is quite equivalent to the number of leaves n_l, since every vertex of the reference mesh at full resolution is a leaf in the forest, we have that the view-dependent tree stores $8n$ bytes (where for every label we assume the cost of 4 bytes). For example, for a mesh with 10.000.000 of vertices, we have that the forest of the view-dependent tree occupies about 80 Mbytes of memory.

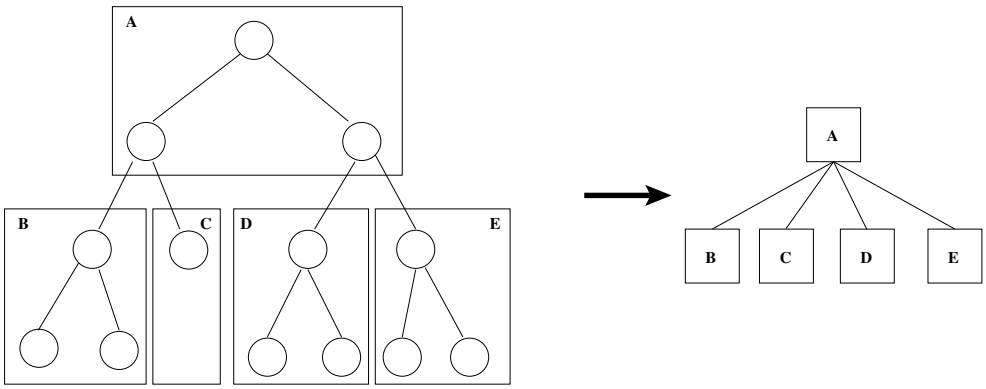


Figure 6.2: Example of view-dependent tree clustering: on the left the tree clustered in blocks, on the right the resulting *meta-tree*.

Thus, the idea is to cluster every binary tree, building a *meta-tree* of *blocks*, where every block groups different levels of each tree. Following El-Sana and Chiang [ESC00], we have chosen to cluster each tree by grouping a fixed number a levels l, trying to determine l in a way that every block can be packed and easily transmitted through the network. Every block is identified by a label and contains a binary tree that is a part of the original tree.

The leaves of every internal block have links to the other blocks which have as root their sons in the view-dependent tree.

The resulting meta-tree is not a binary tree, but it is a n-ary tree, with the number of nodes for every block bounded by the choice of l. The forest of n-ary trees generated from the clustering process has a moderate number of nodes and can be efficiently encoded to be transmitted from the server to the client during the initialization phase. Only the bare nodes of the meta-tree are transmitted, while their content will be transmitted later on, upon queries from the client, during selective refinement. In Figure 6.2 we show an example a meta-tree obtained by grouping nodes of two levels.

6.2.3 The MT on the Server

On the server side, we store the whole data structure, in a similar way as in a classical standalone implementation. The main difference is that, in this case, we must prepare the data to be transmitted. In detail, we have:

- The **base mesh** at the lowest resolution: generically, if the simplification process applied to build the VDMT is quite aggressive, this is not a large mesh, and the time to transmit it in standard format is negligible. However, this mesh is necessary to start to query the VDMT, so it has to be sent entirely to the client. If needed, we can compress the mesh to send it quickly. For a survey on mesh compression techniques, see [Dan05].
- The **vertex forest** that encodes partial dependencies among the update operations: this structure, in the case of a large initial data set, can be quite large too, because it is in the order of the number of vertices of the reference mesh. The idea is to cluster the forest of binary trees in such a way that a single block can be easily transmitted. So, the server has to store the trees clustered into blocks, and a meta-tree that encodes the dependencies among the different blocks.
- The set of **updates**: these are almost the same as in the standalone structure, depending on the context we are treating. We recall that this set is indexed based on the label assigned to each update in the enumeration scheme. Since we have a compact encoding, we assume that we can transmit each update without further compression.

In addition, the server may have to store a restricted amount of information about the client, just to validate each session for security reason. For example, the server has to store the session identifier that it has assigned to each client. Other information can be the ones related to cryptographic keys established with the client to guarantee the security and the privacy of the transmitted data.

6.2.4 The MT on the Client

On the client side, we have to store sufficient information in order to allow the client itself to perform selective refinement queries, possibly asking the server for more updates in order to complete the operation.

Thus, the client stores:

- the **current mesh**;
- the bare **meta-tree** built on the view-dependent tree by the server.

Moreover, the client can store in a cache also the content of some of the blocks of the meta-tree with the corresponding updates.

If the client has sufficient capacity to store the whole VDMT we can refer at the entire process as a *streaming* process. When the client has received from the server the whole forest of trees and all the updates, it does not need to ask the server for more information. Otherwise, if the client has storing limitations, it has to use caching policies to store the content of some blocks of the meta-tree and the related updates. In this case, the client cannot disconnect form the server if the user wants to continue to query the multi-resolution model.

Caching policies on the client Discussing about caching policies on the client side, we have to consider the amount of caching space on the client and to decide some priorities. First of all, we consider that, to perform selective refinement query, the client has to be able to access the view-dependent tree to test the feasibility of update operations. For this reason, the client should be able to cache the content of a reasonable amount of blocks of the meta-tree. Since every block is sent by the server with the corresponding updates, the client has to maintain in the cache as long as possible the updates which labels are those of the cached blocks of the meta-tree.
To replace the content of the cache, the client uses a *least recently used* approach, keeping in cache the blocks of the meta-tree and, possibly, the corresponding updates, which were accessed more recently. Notice that the same policy has to be used also on the cached updates, when the server sends single updated without blocks of the meta-tree.
Since caching policies are very important for selective refinement in this context, in the implementation phase it will be useful to make several experiments to find more precise solutions.

6.2.5 The Communication Session

The communication protocol is essentially based on the exchange between the server and the client of a set of messages, as illustrated in Figure 6.3.

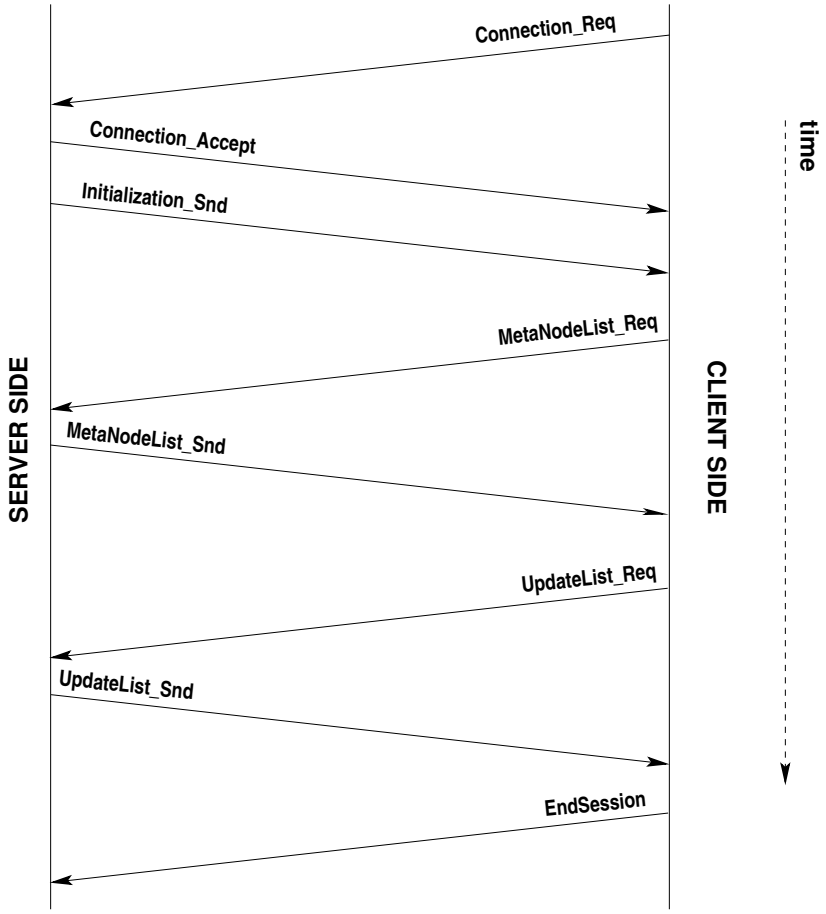


Figure 6.3: An example of a communication session.

The process starts with a request for connection from the client to the server: the client sends to the server a `Connection_Req` message. If the connection can be established, the server replies with a `Connection_Accept` message, in which it specifies the session identifier and, possibly, other privacy and security informations, such as, for example, a cryptographic key for protection of data.

The server first sends to the client the base mesh and the bare meta-tree, which describes the dependencies among the block of the clustered forest of trees. These data are sent with the `Initialization_Snd` message. When the server has completed the transmission of the initialization data, it enters a listening status, waiting for requests from the client.

The client, to perform selective refinement according to user request of a mesh at a desired level of detail, can generate two kinds of request messages. This messages are:

(i) `MetaNodeList_Req`, that specify to the server a request for a list of blocks of the meta-tree;

(ii) `UpdateList_Req`, that specify to the server a request for a subset of the encoded updates.

When the server receives one of these messages, it seeks for the requested data in the VDMT and prepares the packets to be transmitted to the client. Respectively, the reply messages are:

(i) `MetaNodeList_Snd`, which contains, for every block requested from the client, the content of the block and the corresponding updates. These updates are those labeled with the same labels of the part of the tree in the selected block.

(ii) `UpdateList_Snd`, which contains the list of updates requested from the client.

The connection goes on with the exchange of these two couple of messages, a request and a subsequent reply, until either the client has reconstructed all the VDMT, or user stop the process because of other reasons. To end the session, the client sends to the server an `EndSession` message: when the server receives this messages, it simply close the communication channel and discards the informations for the identification of the client that it has stored.

6.3 Selective Refinement

As we have seen, the basic operation on an LOD model, selective refinement, consists of extracting a mesh satisfying some application-dependent requirements based on level of detail, such a request from a user to view the scene much closer or from an other point of view.

We refer here to the primitives we have introduced in Section 5.3, that are sufficient to implement an incremental selective refinement algorithm on an HET. This set of primitives is designed for the HET data structure, but they are very close to the primitives defined for the NMT, except obviously for the part the apply the modifications to the mesh. We present here the design for the HET. The selective refinement primitives we refer are defined as follows:

- Primitive `Collapse` performs a simplification operation (corresponding to an update u) on the current mesh.
- Primitive `Split` performs a feasible refinement operation (corresponding to an update u) on the current mesh.
- Primitive `CollapseTest` checks the feasibility of a simplification operation on the current mesh.
- Primitive `SplitTest` checks the feasibility of a refinement operation on the current mesh;
- Primitive `ForceSplit` performs a split on the current mesh, after having recursively performed all the splits needed to achieve its feasibility.

In the network framework we are considering, we are not interested to the primitives that actually update the mesh, because we can think to this task as a *black box*: the client receives from the server the blocks of the meta-tree and the corresponding modifications, and, once decided what modifications are useful to fulfill the user request, simply calls the appropriate primitive, according to the context, tetrahedra of triangle-segment meshes.

The more interesting primitives in the client/server case, are those performing the *feasibility test* of a single modification operation, following the rules recalled in Section 5.2.1 or in Section 4.4.2.

For the primitive **SplitTest** we do not have problems: the splitting rule states that, given a vertex w as the vertex split by modification u, w can be split if all modifications u', such that $u' \prec u$, are already in S. That means, that each vertex v adjacent to w has a label lower than w. In this case, the vertices adjacent to w belongs all to the current mesh, so the client has all the labels and can perform the feasibility test. If the test fails, the **SplitTest** primitive returns the index of a vertex that has to be split to fulfill the condition. This operation is done from the primitive **ForceSplit**. Once the splitting test is completed, the client has a orderer set of modifications to apply to the mesh: if it has already in cache, there is no problems, if not, it must send a request to the server specifying the index of the modifications it needs.

In the other test, the **CollapseTest** primitive, the rule we have to follow states: 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 parent (or true parent for the HET) with a label greater than label of w', or no true parent (i.e. w' is a root in the NMT). In this case, the client performs the check by using the forest of binary trees, and not only the labels it has on the current mesh. So every time the client needs to check labels belonging to blocks of the meta-tree that it does not have in cache, it

have to request to the server for the block it suppose it needs. Once the test is completed, it the modification is feasible, the client can perform it.

Chapter 7

Conclusion and Future Work

Contribution of this thesis is in the field of representation of two- and three-dimensional complex shapes through efficient multi-resolution data structures. Modern technologies have lead to generation of large, and sometimes huge, datasets, with hundred of thousand of points. Multi-resolution (or Level-of-Detail (LOD)) models provide a powerful representation, giving to users the possibility of extract from a large model a mesh with variable level of detail in different parts of the domain, without visualize the entire dataset at high resolution. By designing compact and efficient data structures for multi-resolution models, we can represent and selectively refine also large meshes.

In this thesis we describe efficient data structures defined as specialization in different application fields of the general multi-resolution model called *Multi-Tessellation*. With the term efficient we refer to data structures with two main characteristics: a compact encoding of the modification operation needed to modify the raw base mesh, and an efficient representation for the dependency relation among that operations. For this second issue, we have chosen to follow [ESV99] and use the *view dependent tree*.

Thus, we have designed and implemented multi-resolution data structures in two different contexts: the representation of non-regular non-manifold two-dimensional objects embedded in three-dimensional fields, and the representation of three dimensional scalar fields through tetrahedral meshes.

For the first field, we have described the *Non-Manifold Multi-Tessellation (NMT)*, an extension to the non-manifold case of the standard MT model. We have designed and implemented a compact data structure for the NMT, using a compact encoding of the modifications and the view-dependent tree for the dependency relation. We have also proposed a new data structure for non-regular non-manifold mesh, called the *Triangle Segment*, which provides a powerful way to represent and manipulate this kind of meshes. For future research, an interesting open problem is the generalization of the explicit Multi-Tessellation

to the non-manifold domain, studying also the interoperability with the Triangle Segment data structure.

For three-dimensional scalar field we have proposed the *Half-Edge Tree*, a compact multi-resolution model built through *half-edge collapse*. In this case, for the dependency relation, we have modified the view-dependent tree for a correct encoding of the dependencies generated by the sequence of half-edge collapse operations. In Chapter 5 we have also made a comparison with the *Full-Edge Tree*, a data structure based on *full-edge collapse*.

Another relevant issue is transmission and remote manipulation of multi-resolution models over the networks. The growing availability of wide band access has made easier the transmission of geometric data: the interesting issue here, is the design of systems which permit to remote clients to visualize and selectively refine a multi-resolution model stored on a centralized server. In this thesis we have proposed a preliminary design of a client/server framework for the class of MTs based on view-dependent tree, that we have called *View-Dependent MT (VDMT)*.

The client/server framework we have described in this thesis is a preliminary version: the first aim is the implementation, which, through experimental approach, can lead to a more precise definition of the system. Further research directions are:

(a) the design and the implementation of a client/server system for the explicit Multi-Tessellation data structure; we are currently facing this problem because the MT is the interface for the shape repository of the European Network of Excellence AIM@SHAPE;

(b) the implementation of the client/server framework designed in this thesis, and the design and the implementation of an extension for the World Wide Web context;

(c) the study of applications of this approach to other compact representations for the MT, to possibly define a general framework to transmit and remote query a MT model.

Bibliography

[AD01] Pierre Alliez and Mathieu Desbrun. Progressive compression for lossless transmission of triangle meshes. In *SIGGRAPH 2001 Conference Proceedings*, pages 198–205, 2001.

[Bey95] J. Bey. Tetrahedral mesh refinement. *Computing*, 55:355–378, 1995.

[BK02] S. Bischoff and L. Kobbelt. Towards robust broadcasting of geometry data. *Computers & Graphics*, 26(5):665–675, 2002.

[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.

[CA95] W. Charlesworth and D. C. Anderson. Applications on non-manifold topology. In *Proceedings Computers in Engineering Conference and Engineering Database Symposium*, pages 103–112. ASME, 1995.

[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.

[CDFM+00] P. Cignoni, L. De Floriani, P. Magillo, E. Puppo, and R. Scopigno. Volume visualization of large tetrahedral meshes on low cost platforms. In *Proceedings NSF/DoE Lake Tahoe Workshop on Hierarchical Approximation and Geometrical Methods for Scientific Visualization*, October 2000.

[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.

[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.

[Dan05] E. Danovaro. *Multi-resolution Modeling of Discrete Scalar Fields*. PhD thesis, Dept. of Computer and Information Sciences, University of Genova (Italy), 2005.

[DDFMP01] 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.

[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.

[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.

[DFM+05a] E. Danovaro, L. De Floriani, P. Magillo, E. Puppo, and D. Sobrero. Level-of-detail for data analysis: a historical overview and some new perspectives. *Computer & Graphics*, 2005. Accepted for publication.

[DFM+05b] 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 2005 (SMI' 05)*, pages 334–339. IEEE Computer Society, to appear - 2005.

[DFM+06] E. Danovaro, L. De Floriani, P. Magillo, E. Puppo, and D. Sobrero. Multiresolution modeling on a client-server architecture. Technical report, Department of Computer and Information Science, University of Genova, Genova (Italy), 2006. Technical report SPADA@WEB.

[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.

[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.

[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)*, pages 159–170. ACM Press, June 2002.

[DFMPS04] L. De Floriani, P. Magillo, E. Puppo, and D. Sobrero. A multi-resolution topological representation for non-manifold meshes. *Computer-Aided Design Journal*, 36(2):141–159, February 2004.

[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.

[DH05] L. De Floriani and A. Hui. Data structures for simplicial complexes: an analysis and a comparison. In M. Desbrun and H. Pottmann, editors, *Third Eurographics Symposium on Geometry Processing*, pages 119–128, Vienna, Austria, July 4-6, 2005.

[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.

[EKT01] W. Evans, D. Kirkpatrick, and G. Townsend. Right-triangulated irregular networks. *Algorithmica*, 30(2):264–286, 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.

[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.

[ESV99] J. El-Sana and A. Varshney. Generalized view-dependent simplification. *Computer Graphics Forum*, 18(3):C83–C94, 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.

[GH97] M. Garland and P. S. Heckbert. Surface simplification using quadric error metrics. In *Computer Graphics Proceedings, Annual Conference Series (SIGGRAPH '97)*, pages 209–216. ACM Press, 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.

[GR99] T. Gerstner and M. Rumpf. Multiresolutional parallel isosurface extraction based on tetrahedral bisection. In *Proceedings 1999 Symposium on Volume Visualization*. ACM Press, 1999.

[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.

[Hop98] H. Hoppe. Efficient implementation of progressive meshes. *Computer & Graphics*, 22(1):27–36, 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 *International Conference on Shape Modeling and Applications (SMI 2004)*, pages 209–220. IEEE Computer Society, 2004.

[Kob00] L. Kobbelt. $\sqrt{3}$ subdivision. In *Proceedings of ACM SIGGRAPH 2000*, pages 103–112. ACM, 2000.

[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.

[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. ACM Press, June 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.

[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.

[Man88] M. Mantyla. *An Introduction to Solid Modeling*. Computer Science Press, 1988.

[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.

[McM00] S. McMains. *Geometric Algorithms and Data Representation for Solid Freeform Fabrication*. PhD thesis, University of California at Berkeley, 2000.

[oEA] European Network of Excellence AIM@SHAPE. www.aimatshape.net.

[OR97] M. Ohlberger and M. Rumpf. Hierarchical and adaptive visualization on nested grids. *Computing*, 56(4):365–385, 1997.

[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.

[Pas02] V. Pascucci. Slow Growing Subdivisions (SGSs) in any dimension: towards removing the curse of dimensionality. *Computer Graphics Forum*, 21(3), 2002.

[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.

[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.

[Pup98] E. Puppo. Variable resolution triangulations. *Computational Geometry Theory and Applications*, 11(3–4):219–238, 1998.

[RL92] M. Rivara and C. Levin. A 3D refinement algorithm for adaptive and multigrid techniques. *Communications in Applied Numerical Methods*, 8:281–290, 1992.

[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.

[Sam05] H. Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan-Kaufmann, 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.

[SP92] L. Scarlatos and T. Pavlidis. Hierarchical triangulation using cartographics coherence. *CVGIP: Graphical Models and Image Processing*, 54(2):147–161, March 1992.

[SWH95] R. Sriram, A. Wong, and L. He. Gnomes: an object oriented non-manifold geometric engine. *Computer Aided Design*, 27(11), 1995.

[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.

[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.

[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.

[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.

[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.