A Dimension-Independent Library for Building and Manipulating Multiresolution Triangulations

Leila De Floriani†, Paola Magillo, Enrico Puppo

Dipartimento di Informatica e Scienze dell'Informazione – Università di Genova
Via Dodecaneso, 35, 16146 Genova, ITALY
Email: {deflo,magillo,puppo}@disi.unige.it

Abstract

A Multi-Triangulation *(MT) is a general multiresolution model for representing k-dimensional geometric objects through simplicial complexes. An MT integrates several alternative representations of an object, and provides simple methods for handling representations at variable resolution efficiently , thus offering a basis for the development of applications that need to manage the level-of-detail of complex objects. In this paper, we present an object-oriented library that provides an open-ended tool for building and manipulating object representations based on the MT.*

1. Introduction

Geometric cell complexes (meshes) have a well-established role as discrete models of continuous domains and spatial objects in a variety of application fields, including virtual reality, scientific visualization, computer aided design, Geographic Information Systems (GISs), etc. In particular, simplicial complexes (e.g., triangle and tetrahedra meshes) offer advantageous features in terms of adaptivity to the shape of the entity as well as ease of manipulation.

The accuracy of the representation achieved by a discrete geometric model is somehow related to its resolution, i.e., to the relative size and number of its cells. At the state-of-the-art, while the availability of data sets of larger and larger size allows building models at higher and higher resolution, the computing power and the transmission bandwidth of networks are still insufficient to manage such models at their full resolution. The need to trade-off between representation accuracy, and time and space constraints imposed by applications has motivated a burst of research on *Level-of-Detail (LOD)*. The general idea behind LOD can be summarized as: *always use the maximum resolution you need – or you can afford – and never use more than that.* In order to apply this principle, a mechanism is necessary that can "administrate" resolution, by adapting a mesh to the needs of an application, possibly varying its resolution over different zones of the entity represented.

A number of different LOD models have been proposed in the literature. Most of them have been developed for applications to terrain modeling in GISs [4, 9, 14], and to surface representation in computer graphics, and virtual reality [15, 12, 18, 11]. Such models are strongly characterized by the data structures and optimization techniques they adopt, as well as custom tailored to perform specific operations. In this scenario, developers who would like to include LOD features in their applications are forced to implement their own models, and mechanisms. On the other hand, a wide range of potential applications for LOD have been devised, which require a common basis of operations [8]. Therefore, it seems desirable that the LOD technology is brought to a more mature stage, which allows developers to use it through a common interface, without the need to care about too many details.

In our previous work, we have developed a general model, called a *Multi-Triangulation (MT)*, that can capture all LOD models based on simplicial complexes as special cases [17, 10] (here, we use *triangulation* as a generic term to denote a simplicial complex in any dimension). Based on such model, we have built systems for managing the level of detail in terrains [6], and in free-form surfaces [8], and we are currently developing an application in volume visualization.

Here, we present a library for building and manipulating MTs, which provides an open-ended tool for the de-

† On leave from the University of Genova at the University of Maryland Institute for Advanced Computer Studies (UMIACS).

© The Eurographics Association and Blackwell Publishers 1999. Published by Blackwell Publishers, 108 Cowley Road, Oxford OX4 1JF, UK and 350 Main Street, Malden, MA 02148, USA.

velopment of applications that need advanced LOD features. Although most models proposed in the literature deal just with two-dimensional complexes (e.g., surface meshes), our framework is dimension independent, hence suitable to handle also volume, and higher dimensional data. The definition of MT, its properties, and the techniques to manipulate it, as well as the whole software offered by the library, are defined in a unique way and parametrically on the dimension of basic cells.

The reminder of this paper is organized as follows: in Section 2 we briefly review the Multi-Triangulation; in Section 3 we present the overall organization of the library; in Sections 4 and 5 we describe operations for constructing and querying an MT, respectively; in Section 6 we discuss some implementation details; finally, Section 7 contains some concluding remarks.

2. The Multi-Triangulation

In this section we briefly review main concepts about the Multi-Triangulation [17, 10]. For the sake of brevity, this subject is treated informally here. For a complete and formal treatment see [16].

A *k-dimensional simplex* σ is the locus of points that can be expressed as the convex combination of $k + 1$ affinely independent points in $\mathbb{R}^d$, called the *vertices* of σ. Any simplex whose vertices are a subset of the vertices of σ is called a *facet* of σ. A *k-dimensional simplicial complex* in $\mathbb{R}^d$ is a finite set Σ of k-simplices such that, for any pair of distinct simplices $\sigma_1, \sigma_2 \in \Sigma$, either σ_1 and σ_2 are disjoint, or their intersection is the set of facets shared by σ_1 and σ_2. In what follows, a k-simplex will be always called a *cell*.

The intuitive idea behind a *Multi-Triangulation* (MT) is the following: consider a process that starts with a coarse simplicial complex and progressively refines it by performing a sequence of local updates. Each *local update* replaces a group of cells with another group of cells at higher resolution. Figure 1 shows a sequence of updates performed on a triangle mesh. An update C_2 in the sequence *directly depends* on another update C_1 preceding it if and only if C_2 removes some cells introduced with C_1. The *dependency relation* between updates is defined as the transitive closure of the direct dependency relation. Only updates that depend on each other need to be performed in the given order; mutually independent updates can be performed in an arbitary order. For instance, in the example of Figure 1, updates 3 and 4 are mutually independent, while update 5 depends on both; thus, we can perform updates 3 and 4 in any order, but we must perform them both before we can perform update 5.

An MT abstracts from the totally ordered sequence by encoding a *partial order* describing the *mutual dependencies* between pairs of updates. Updates forming any subset closed with respect to the partial order, when performed in a consistent sequence, generate a valid simplicial complex. It is possible to perform more updates in some areas and fewer updates elsewhere, thus producing a complex whose resolution is variable in space. Such an operation is known as a *selective refinement*, and it is at the basis of LOD management. Results on selective refinement from an MT representing a terrain are shown in Figure 2.

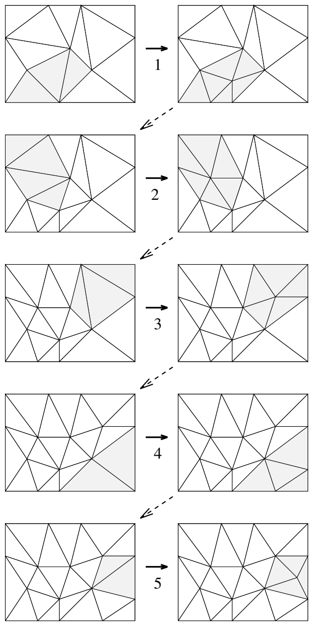
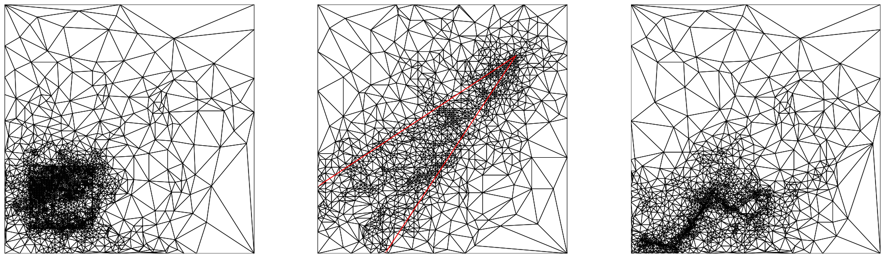


Figure 1: *A sequence of five updates (numbered 1. . . 5) progressively refining an initial coarse triangle mesh. The area affected by each update is shaded.*

An MT is described by a directed acyclic graph (DAG). Each update is a node of the DAG, while the arcs correspond to direct dependencies between updates. Each arc is labeled with the collection of all cells of its source node that are removed by its destination node. For convenience, we introduce two further nodes: a *root*, corresponding to the initial coarse complex, which is connected with an arc to each update that removes some if its cells; and a *drain*, corresponding to the complex obtained by performing all updates, which is connected through an arc outgoing from each update that creates some of its cells. Such arcs are labeled by cells in a consistent way. Figure 3 shows the MT corrsponding to the collection of updates described in Figure 1.

A *front* of an MT is a set of arcs containing exactly one arc on each directed path from the root, as shown in Figure 3. Nodes lying before a front form a consistent set of updates; the corresponding simplicial complex is formed by all cells labeling the arcs of the front [16]. By sweeping a front through the DAG, a wide range of complexes is generated, each characterized by a different resolution, possibly variable in space.



(a) (b) (c)

Figure 2: *Three meshes extracted from a two-dimensional MT representing a terrain (top view). (a) The triangulation has the highest possible resolution inside a rectangular window, and the lowest possible resolution outside it. (b) Resolution inside a view frustum (wedge) is decreasing with the distance from its focus point, while it is arbitrarily low outside it. (c) Resolution is high only in the proximity of a polyline.*

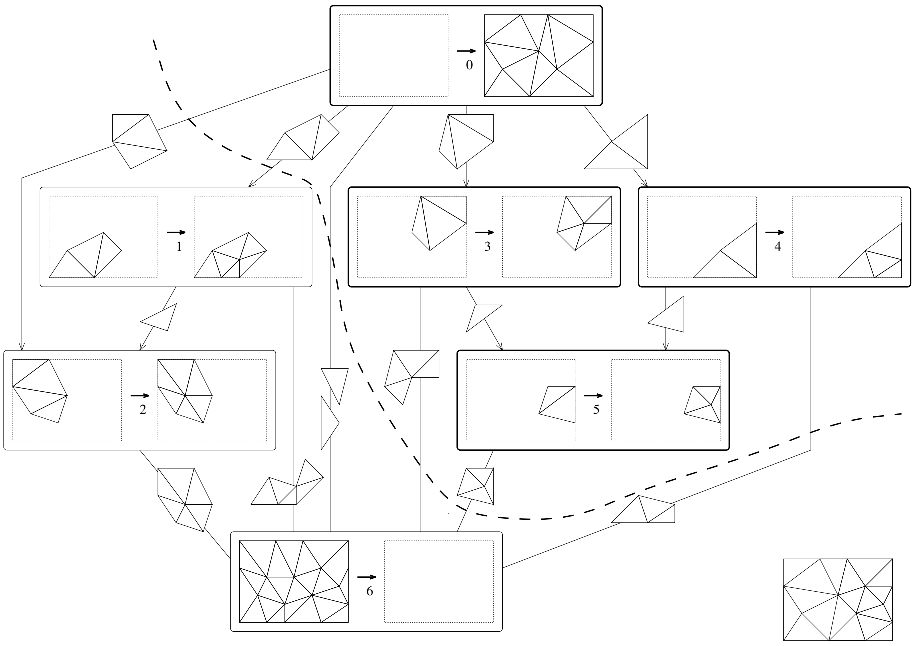


Figure 3: *The MT built over the partially ordered set of mesh updates of Figure 1. Each node represents a mesh update and shows the two sets of cells removed and created in the update. Each arc represents the dependency between two updates and is labelled with the cells created in the first update which are removed in the second update. A front on the MT contains the arcs intersected by the dashed line; nodes lying before the front are highlighted. The mesh associated with the front is shown on the right.*

2.1. Building an MT

An MT can be built from of any sequence of local updates produced by an algorithm for *mesh refinement*, or *mesh simplification*.

A refinement algorithm starts from a mesh at low resolution, and incrementally produces meshes at higher resolution. Mesh refinement has been widely used in the context of terrain approximation [13], and in multiresolution volume visualization [2]. Most of such algorithms are based on the iterative application of local updates characterized by a specific pattern. A common pattern is *vertex insertion*, as depicted in Figure 4 for the two-dimensional case: an initial coarse mesh is built on a small set of vertices, while other vertices are inserted one at a time; each time a new vertex is inserted, a polytope formed by a set of adjacent cells in the current mesh is retriangulated through a new set of cells incident at the new vertex. An MT can be built top-down by first setting its root at the initial mesh, and then tracing the sequence of updates performed during mesh refinement: each update defines a new node, which is linked to the DAG according to its interferences with the existing nodes.

An algorithm for mesh simplification starts from a mesh at high resolution, and incrementally produces meshes at lower resolution by decreasing the number of

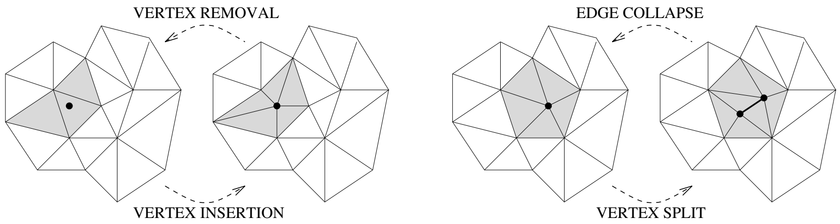


Figure 4: *The two types of update patterns used on two-dimensional meshes in refinement (left-to-right), and in simplification (right-to-left). The shaded triangles are those involved in the update. Such patterns extend directly to a generic k-dimensional case.*

cells in the input mesh. A large number of simplification algorithms have been developed for different applications [3]. Also simplification can be performed by iteratively applying local updates: this process is similar to rolling back a refinement sequence. Two common patterns for simplification updates are *vertex decimation*, and *edge collapse*, as shown in Figure 4 in the two dimensional case. An MT can be built bottom-up by setting its drain at the initial mesh, and making a new node at each local update, as for refinement. The only difference is that the two sets of cells involved in an update are swapped in the MT node, which becomes an ancestor, rather than a descendant of the nodes generated by previous updates.

Thus, a *history file* that traces local updates performed through a generic iterative refinement or simplification process contains sufficient information for building an MT. The key operations consist of building a node from a local update and of inserting it consistently in the DAG representing the MT.

In our library, we provide general and dimension-independent mechanisms for building an MT starting from a history file, as well as some refinement and simplification algorithms that we have implemented for the two- and three-dimensional cases. Such issues will be discussed in Section 2.1.

2.2. Querying an MT

Since an MT provides several descriptions of a spatial object by tracing interferences among different local approximations, there are two general approaches in using this model:

- As an efficient mechanism to obtain a representation of the whole object according to some user-defined resolution requirements, where resolution is possibly variable over different parts of the object;
- As a spatial indexing scheme to query a region of interest. Also in this case, the resolution of the portion of object returned may depend on user-defined requirements.

In [8], we have proposed general techniques to traverse an MT and to extract either a portion or the whole object, at a user-defined resolution. Such techniques have been included in our library. They will be summarized in Section 2.2. A detailed description is given in [16].

Queries on an MT are defined through *regions of interest*, and *resolution filters*. A *region of interest (ROI)* is a generic, finitely describable subset of the space $\mathbb{R}^d$ in which the complex is embedded. Common choices are a point, a ray, a straight-line segment, a hyperplane, a half-space, a (hyper)box, a (hyper)ball, etc., as well as any finite union or intersection of such objects. A cell of an MT is said *active* with respect to some ROI if and only if their intersection is not empty. In our library, we provide some methods to test cells with respect to some frequently used ROIs as well as general mechanisms to include user-defined ROIs.

A *resolution filter* is a user-defined function R that assigns to each cell σ of the MT a real value $R(\sigma)$. Intuitively, a resolution filter measures the "signed difference" between the resolution of a cell, and that required by the application: $R(\sigma) > 0$ means that the resolution of σ is not sufficient; $R(\sigma) < 0$ means that the resolution of σ is higher than necessary. A cell such that $R(\sigma) \leq 0$ is said *feasible*.

The value of the resolution filter at a cell σ may depend on the position of σ in space, on its attributes, on its geometric properties (size, shape, ecc.), as well as on other user-defined parameters. Regions of interest can be combined with analytic functions in the definition of resolution filters, i.e., a filter may be defined by a function for a cell that is active with respect to some ROI, by a different function for another cell that is active for a different ROI, and so on.

For example, the meshes depicted in Figure 2 satisfy the following resolution filters: in (a) R is negative for all cells outside the window, zero for all cells inside it that are at the highest resolution, and positive for all others; in (b) R is negative for all cells outside the view frustum, while for a cell σ inside it, R is decreasing with resolution of σ, and with its distance from the focus point; in (c) R is negative for all cells not intersecting the polyline, zero for all cells intersecting it that are at the highest resolution, and positive for all others.

We provide the following three traversal algorithms for the MT:

- A *static* algorithm, which is specified by an MT and a resolution filter. This algorithm returns the simplicial complex of minimum size (i.e., composed of the smallest number of cells) which satisfies the resolution filter. It is suitable to single queries about a representation of the whole object.
- A *dynamic* algorithm, which is specified by an MT, a front on it, and a resolution filter. This algorithm returns the complex that is as close as possible to the given front and is minimally sufficient with respect to the resolution filter (i.e., it cannot be simplified further without violating it). It is suitable to repeated queries, when the result of a new query differs just slightly from the result of the previous one (e.g., in a virtual reality environment where the resolution filter depends on the position and on the viewing direction of a moving point).
- A *local* algorithm, which is specified by an MT, a resolution filter, and a ROI. This algorithm returns a representation of the portion of the object inside the ROI, which satisfies the resolution filter, and cannot be simplified without violating it. This is suitable to local queries: depending on the ROI used, it implements specific geometric queries, like point location, windowing, and ray casting.

Variants of these algorithms are also described in [16], in which the condition on the output complex of being formed by valid cells is relaxed, while a given bound on its size is imposed. In this case, each algorithm returns the complex within the given bound that is as close as possible to that satisfying the resolution filter.

Mesh extraction algorithms will be discussed in Section 5.2.

3. Library Overview

Our major objective in designing and developing a library for the manipulation of Multi-Triangulations is to provide an *open system* which can be integrated with user modules. Such a system permits to build applications to perform analysis, processing, and visualization of geometric models in any dimension, with the fundamental characteristic of using resolution as a further parameter in the computation.

3.1. Library Architecture

The system architecture consists of *library modules* and *user modules*. Library modules are the "standard" part of the system; they provide operations that allow a user to build and query an MT. User modules are the extensible part of the system; they include programs which generate histories for building MTs and programs that inquiry an MT to perform specific operations at variable resolution. In addition, special user modules manage the application-dependent attributes of cells and vertices of a Multi-Triangulation.

The library modules are:

1. The *MT-manager*: this is the only module having direct access to the MT data structure; it provides a standard interface for the MT and contains:
   - *query operations* described in Section 2.2 (static, dynamic and local algorithms), which represent the interface to user modules;
   - *construction operations* used to build an MT from a given history file, which represent the interface to the MT-builder (see below);
   - *I/O operations* used to read/write an MT from/to disk.
2. The *MT-builder*: this module builds an MT starting from a history file.
3. The *MT-history-tracer*: this module provides utilities that allow a user program to write a history in the correct format to be read by the MT-builder. Its primitives can be called within any user program incrementally refining or simplifying a mesh in order to trace the history of the iterative process on disk.
4. A generic module *MT-query-params*, which provides the shallow interface of resolution filters and ROIs to be used as parameters to query an MT through the MT-manager operations; user-defined resolution filters and ROIs reduce to specific implementations of such interface.
5. A generic module *MT-attribute-manager*, providing the shallow interface for handling the application- dependent attributes of cells and vertices; the user defines its attribute managers as specific implementations of such interface.

Library modules are parametric on the size d of the space containing data and on the dimension k of the triangulation. User-defined MT-attribute-managers can be associated with modules 1–4 in order to obtain specialized MT-managers, MT-builders, MT-history-tracers and MT-query-params which operate on MTs having some specific attributes associated with their cells and/or vertices. This structure allows a user to adapt the library to her/his specific needs while maintaining all general mechanisms unchanged.

User modules can be subdivided into the following types:

1. *MT-history-generators*, which implement methods for the iterative refinement or simplification of a mesh, and use the primitives the MT-history-tracer to write the history of the process.
2. *MT-clients*, which use the query operation of the MT-manager to perform operations at variable resolution on geometric objects encoded through an MT.
3. *Resolution filters* and *ROIs*, implemented according with the interface of MT-query-params.
4. Modules handling *cells and vertex attributes*, implemented according to the interface of MT-attribute-manager.

The system architecture is depicted in Figure 5. Referring to such Figure, we illustrate two typical scenarios:

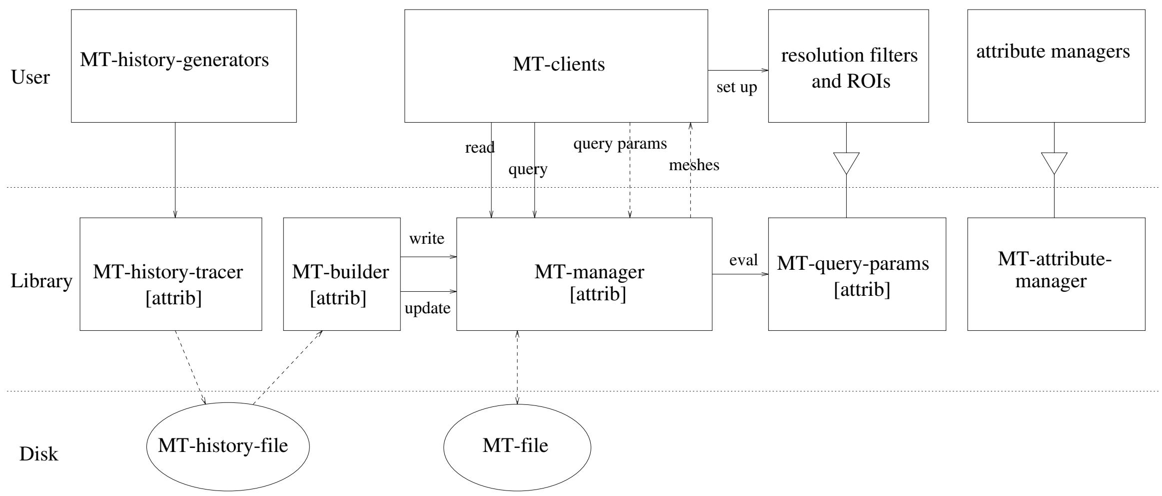


Figure 5: *Components of the MT system and their relations. Dashed arrows represent data flow; solid arrows represent control flow (arrows directed from the module invoking an operation to the module providing such operation).*

the construction of an MT and the extraction of a mesh from an MT.

In order to construct an MT, first we run an MT-history-generator. Such program performs a sequence of updates either refining or simplifying a mesh, and uses the primitives of the MT-history-tracer to write the corresponding history to disk into an *MT-history-file*. Then, the MT-builder reads the MT-history-file, builds an MT from the history, and writes it on disk into an *MT-file*. The MT-history-generators and the MT-builder run off-line. This two-step is provided to allow a user to build an MT through her/his own algorithms without any need to know details about the MT.

An existing MT can be read from disk, and inquired to perform operations on a mesh representing the target object at a user-defined resolution. An MT-client first asks the MT-manager to read an MT from an MT-file, then it sets up the desired query parameters (resolution filters and ROIs), and invokes a query. As a result, the MT-manager returns a mesh.

3.2. Basic Data Types

A Multi-Triangulation is characterized by the following elements:

- the *vertices* involved in the model, with their coordinates in space;
- the *cells* involved in the model;
- the *nodes*, which represent *updates*;
- the *arcs*, which represent *relations of direct dependency* between nodes.

Relevant relations connecting these four types of elements are: the Cell-Vertex (CV) relation, which provides the $k+1$ vertices of a cell; the Cell-Arc (CA) relation, providing the arc that contains a given cell; the Arc-Cell (AC) relation, that gives the set of cells labelling an arc;

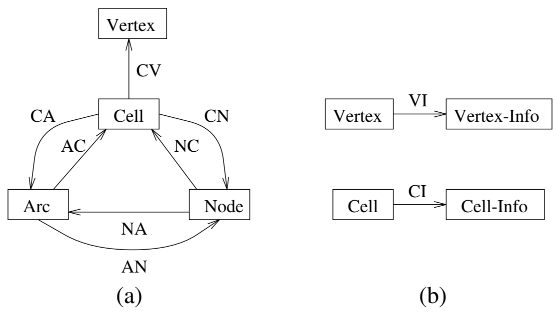


Figure 6: *(a) The relations between the basic elements of an MT; (b) the relations between cells and vertices and their attributes.*

the Cell-Node (CN) relation, that provides the two nodes creating and removing a cell, respectively; and the Node-Cell (NC) relation, that gives the two sets of cells removed and created, respectively, by a node; the Arc-Node (AN) relation, that provides the two nodes connected by an arc; and the Node-Arc (NA) relation, that gives the two sets of arcs entering and leaving a node, respectively. The above relations are shown in Figure 6 (a).

The four basic types and their relations are handled by the MT-Manager which provides the interface operations to the other modules.

3.3. Attributes

Within an application, user-defined attributes (e.g., approximation errors, normals, colors, material properties, textures, field equations, etc.) can be attached to each vertex and to each cell.

Two generic types of elements are provided in the MT-attribute-manager: *vertex attributes* and *cell attributes*.

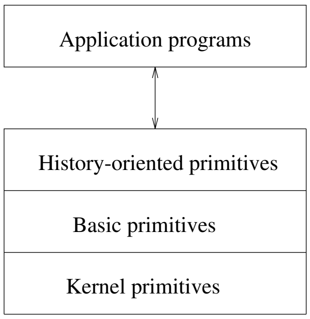


Figure 7: *The three levels of construction operations.*

The relevant relations are the Vertex-Information (VI), and the Cell-Information (CI) relation, which provide the attributes of a given vertex, and of a given cell, respectively (see Figure 6 (b)).

The interface of the MT-attribute-manager contains primitives for managing generic vertex and cell attributes, and relations VI, CI. Depending on the attributes that are of interest for the application, the user can define his/her own instances of attribute managers by providing the specific data structures to contain them and by implementing the interface primitives of the MT-Attribute-Managers.

4. Construction Operations

Operations needed to build an MT are divided into three levels:

1. *Kernel operations*, i.e., basic primitive operations acting on the structure of an MT. Such operations are internal operations of the MT manager and their purpose is hiding the data structure used to represent the MT.
2. *Basic construction operations*, which allow initializing an MT and modifying it according to a new refinement or simplification update. Such operations are internal operations of the MT-builder and allow incrementally building an MT.
3. *History-oriented construction operations*, which directly build an MT starting from a given history. These operations are the interface of the MT-Builder.

Figure 7 shows the relations between the three levels of operations.

4.1. Kernel Operations

The kernel level for MT construction provides operations that modify the internal structure of an MT at the lowest level (i.e., a DAG encoding local updates and their dependency relations). Construction can be performed with the following primitives:

- `AddNode`: adds a new node to the DAG;
- `ConnectCell`: inserts a new cell into an MT, referring it to specified source and destination nodes, and updates relations between such nodes as a consequence (an arc connecting them is created if not already present);
- `DisconnectCell`: removes an existing cell from an MT and updates relations between its source and destination nodes as a consequence (an arc connecting them is deleted if dependency was caused only by the cell deleted).

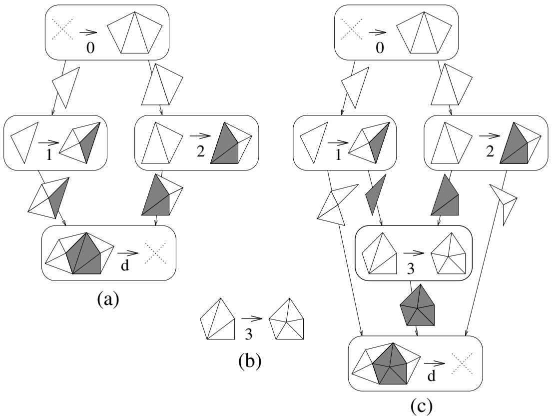


Figure 8: *One step of the top-down construction of an MT: (a) the current MT, (b) the next refinement update to be added to the DAG, and (c) the resulting MT.*

Such operations maintain the invariant property that two nodes are connected by an arc if and only if the destination of the arc removes some cell created by the source.

The kernel provides also operations to evaluate relations described in Section 3.2. Such operations are self-explanatory, and they are not listed here for brevity.

4.2. Basic Construction Operations

Operations in this layer permit to initialize and iteratively add nodes to an MT. Basic construction operations are implemented in the MT-Builder for internal use.

We start with an MT consisting just of the root plus the drain and of one arc connecting them labelled with all cells of the initial cell complex. Then, a new node for each update of the history is created and connected to the DAG based on its dependency relations with the existing nodes.

In construction through refinement, every new update refines an existing MT as follows: a new node corresponding to the update is inserted, which becomes a parent of the drain; some nodes that were parents of the drain become parents of the new node, and they are possibly detatched from the drain. See Figure 8.

In construction through simplification, each new update coarsens the cell complex at the root of the current MT. A new node is inserted, which corresponds to the refinement update obtained by reversing the current update; such node becomes a child of the root; some nodes

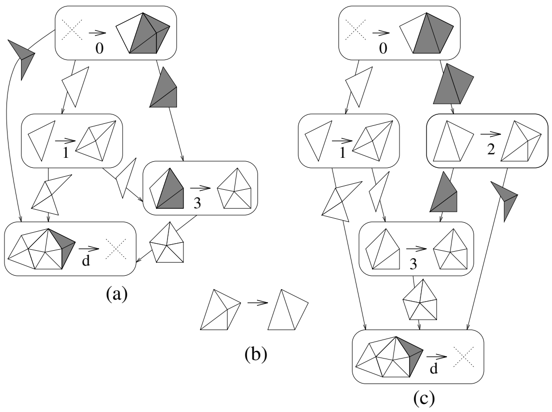


Figure 9: *One step of the bottom-up construction of an MT: (a) the current MT, (b) the next simplification update to be added to the DAG, and (c) the resulting MT.*

$C := \mathcal{M}.\mathtt{AddNode}();$
for each cell $\gamma \in \Gamma_{old}$ do
  $C_i := \mathcal{M}.\mathtt{CellSource}(\gamma);$
  $\mathcal{M}.\mathtt{DisconnectCell}(\gamma);$
  $\mathcal{M}.\mathtt{ConnectCell}(\gamma, C_i, C);$
for each cell $\gamma \in \Gamma_{new}$ do
  $\mathcal{M}.\mathtt{ConnectCell}(\gamma, C, \mathcal{M}.\mathtt{Drain}());$

(a)

$C' := \mathcal{M}.\mathtt{AddNode}();$
for each cell $\gamma \in \Gamma_{old}$ do
  $C'_j := \mathcal{M}.\mathtt{CellDest}(\gamma);$
  $\mathcal{M}.\mathtt{DisconnectCell}(\gamma);$
  $\mathcal{M}.\mathtt{ConnectCell}(\gamma, C', C'_j);$
for each cell $\gamma \in \Gamma_{new}$ do
  $\mathcal{M}.\mathtt{ConnectCell}(\gamma, \mathcal{M}.\mathtt{Root}(), C');$

(b)

Figure 10: *Pseudo-codes of primitives for building an MT through refinement (a), and through simplification (b).*

that were children of the root become children of the new node and are possibly detatched from the root. See Figure 9.

Note that the two methods act on the MT symmetrically, by adding a node immediately before the drain and immediately after the root, respectively.

The basic operations of this level are:

- `InitMC`: given a complex Γ_0, it returns a DAG consisting of just the root node, the drain node, and an arc connecting the root to the drain. The whole set of cells of Γ_0 corresponds to the group inserted by the root and labelling the arc.
- `RefineMC`: given an MT and a refinement update specified by a pair of groups of cells $(\Gamma_{old}, \Gamma_{new})$, it adds a new node representing it and connects such node to the drain and to the nodes that introduced cells of Γ_{old}, possibly detaching them from the drain. See Figure 10a for a pseudo-code of this operation.
- `CoarsenMC` given a simplification update $(\Gamma_{old}, \Gamma_{new})$, it updates the root mesh by applying the update, it add a new node representing its reverse $(\Gamma_{new}, \Gamma_{old})$ and connects such node to the root and to the nodes that introduced cells of Γ_{old}, possibly detaching some of them from the root. See Figure 10b for a pseudo-code of this operation.

4.3. History-Oriented Construction Operations

Operations in this level accept a history and directly build the corresponding MT. There are two operations, corresponding to the construction of an MT top-down (from a refinement history) and bottom-up (from a coarsening history).

- `TopDownMC`: builds an MT from a refinement history. Its implementation reduces to a call to `InitMC` followed by a sequence of calls to `RefineMC`.
- `TopDownMC`: builds an MT from a coarsening history. Its implementation reduces to a call to `InitMC` followed by a sequence of calls to `CoarsenMC`.

5. Query Operations

Operations needed to query an MT are subdivided into three levels (see Figure 11):

1. *Kernel operations*: basic primitive operations acting on the structure of an MT, which hide the specific data structure used to represent an MT, while providing a standard interface to the upper levels.
2. *DAG traversal operations*: defined on top of the kernel and used within the MT-Manager for implementing query operations; they manage the internal state of DAG traversal in the various query algorithms.
3. *Query operations*, which define the user interface of an MT; they implement the three query algorithms outlined in Section 2.2: the static query, the dynamic query, and the local query.

All operations are implemented by the MT-Manager, but only the topmost level (i.e., query operations), and part of the kernel (i.e., operations involving cells, vertices and their attributes) are available to MT-clients.

5.1. Kernel Primitives

This level contain primitives that permit to retrieve the basic relations CV, CA, CN, AC, AN, CN, NA, introduced in Section 3.2. Generic primitives for retrieving relations CI and VI are also provided; the implementation of such primitives relies on some attribute manager that the user must provide when the MT-manager is instantiated; if no

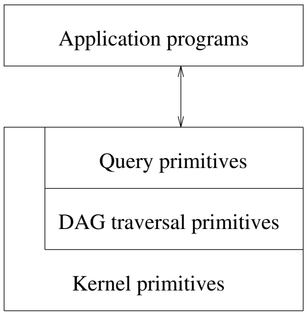


Figure 11: *The three levels of query operations.*

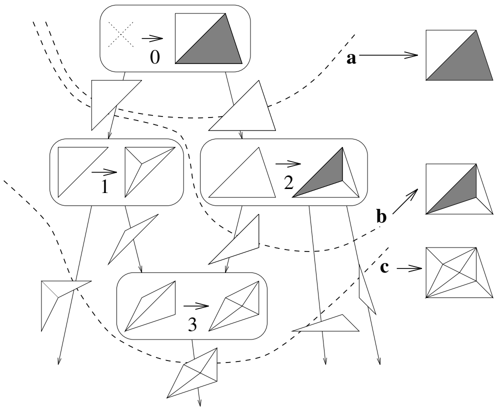


Figure 12: *The static algorithm. Dark triangles are unfeasible.* ***a*** *is the initial front, the corresponding mesh has a dark triangle;* ***b*** *is the current front after sweeping node 2, its mesh still has a dark triangle;* ***c*** *is the final front obtained by sweeping node 3 and its unswept ancestor 1.*

attribute manager is present, then CI and VI simply return null information. In addition, primitives to retrieve the root and the drain of an MT are provided.

Primitives corresponding to relations CV, CI, and VI are available to the user; the others are internal operations of the MT-manager.

5.2. DAG Traversal Primitives

DAG traversal primitives are auxiliary operations acting on the internal state of query algorithms.

The static algorithm (see Figure 12) traverses the DAG representing the given MT by sweeping a front from the root towards the drain. At each step, the current candidate solution is the cell complex associated with the current front. As long as the current complex contains some unfeasible cell σ, the current front is advanced after the node C that removes σ; note that not all arcs entering C may belong to the current front when C is selected: in this case, the front is swept over all unswept ancestors of C as well. When all cells are feasible, then the current complex is returned as the solution.

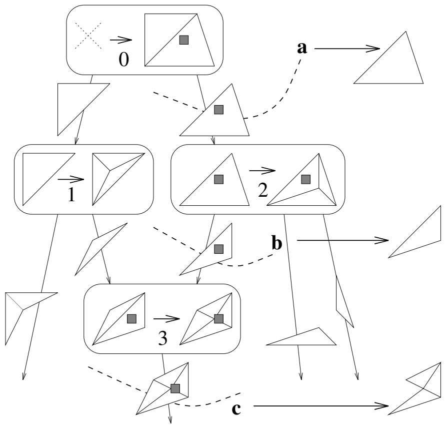


Figure 13: *The local algorithm. The ROI is the dark square.* ***a*** *is the initial front, the corresponding mesh contains one triangle, that we suppose to be not feasible; triangle;* ***b*** *is the current front after sweeping node 2, we suppose that the corresponding triangle is still unfeasible;* ***c*** *is the final front obtained by sweeping node 3; node 1 is not swept since arc (1,3) is not active.*

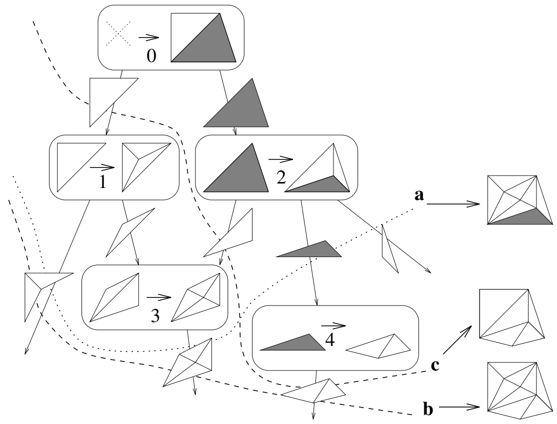


Figure 14: *The dynamicc algorithm. Dark triangles are unfeasible.* ***a*** *is the input front, the corresponding mesh has a dark triangle;* ***b*** *is the current front after sweeping node 4, the mesh has no dark triangles, but it is overrefined;* ***c*** *is the final front obtained by sweeping beyond node 3 and then beyond node 1.*

The local algorithm (see Figure 13) works in a similar way, but it maintains a local front made of *active arcs*, i.e., arcs that have some active cell associated with. Arcs that are not active are ignored. The current candidate solution is given by the collection of the active cells from the arcs of such local front. As long as the current candidate solution contains some unfeasible active cell σ, the algorithm advances the local front after the node C removing σ. Unswept ancestors of C are also swept, if and only if they are connected to C through active arcs.

The dynamic algorithm (see Figure 14) traverses the DAG both upwards and downwards, starting from a given front. First, it applies the same procedure as in the static case to produce a current complex that satisfies the given resolution filter is satisfied on each cell. Then, it coarsens the current complex where its resolution is higher than strictly necessary. by sweeping the current front backwards one node at a time as long as this can be done without introducing unfeasible cells in the current mesh.

The DAG-traversal primitives necessary to implement the static query algorithm are:

- `Init`: sets an initial state in which only the root has been swept and the current front contains all the arcs outgoing from the root.
- `Expansion`(): updates the state by iteratively sweeping the current front forward until all active cells of its arcs are feasible.

The local query algorithm is supported by similar primitives, which have a slightly different behaviors since only active triangles are considered:

- `LocalInit`: sets an initial state in which only the root has been swept and the current local front contains those arcs outgoing from the root which are active.
- `LocalExpansion`(): updates the state by iteratively sweeping the current local front forward until all active cells of its arcs are feasible.

The operations needed for supporting the dynamic algorithm are the `Expansion` operation used in the static case, plus the following:

- `Contraction`: updates the state by sweeping the front before a node as long as this can be done without getting some unfeasible cell in the corresponding mesh.

5.3. Query Operations

The interface of the MT-manager provides to the user operations that correspond to the three query algorithms.

- `StaticQuery`: solves a global query related to a given resolution filter by using the static algorithm; the input parameter is a resolution filter.
- `DynamicQuery`: solves a global query related to a given resolution filter by using the dynamic algorithm; the input parameter is a resolution filter.
- `LocalQuery`: solves a local query related to the current resolution filter and focus set; the input parameters are a resolution filter and a ROI.

The implementation of `StaticQuery` reduces to a call to `Init` followed by a call to `Expansion`; the mesh corresponding to the solution of the query is found as the collection of the cells labelling the arcs of the current front after `Expansion`.

The implementation of `DynamicQuery` reduces to a call to `Contraction` followed by a call to `Expansion`; the internal state inherited from the last query is maintained by the MT-manager in a transparent way, and it is updated in order to compute the answer to the current query.

The implementation of `LocalQuery` is similar to that of `StaticQuery`, but here `LocalInit` and `LocalExpansion` are used; the output mesh is generated by collecting just the active cells of the arcs of the final local front.

6. Implementation Issues

We have been developing a prototype library in C++ under the object-oriented programming paradigm. The dimension-independent library is at an advanced stage of design, while simplified two- and three-dimensional versions are already implemented, which provide basic modules that will be extended to the generic case.

In this section, we first outline the main concepts related to the design of the library, and then we describe software that we already implemented, which will be used to implement the final prototype.

Note: We foresee that the library shall be completed, and released in the public domain, by the time of the conference. The content of this section will be updated consequently in the final paper.

6.1. Object-Oriented Design

The library is organized as a set of classes; such classes and their mutual relationships are represented in Figure 15, and explained in the following.

The MT-manager module is formed by two classes: the *MT* and the *extractor*. Class *MT* stores the vertices, cells, nodes and arcs of an MT, and implements kernel primitives (both for construction and for query), and primitives for disk I/O. Class *MT* must be instantiated by providing the dimension k of its cells, and the dimension d of the embedding space.

The data structure used to represent an MT consists of four arrays storing the vertices, cells, nodes and arcs of the DAG, respectively. A minimal complete subset of the relations between these entities is explicitly stored, namely relations CV, CA, AC, AN, NA; all other relations are found on-the-fly by combining the stored ones in optimal time. Optimized data structures for MTs where all updates are based on a specific rule have also been designed, which represent cells in an implicit way leading to consistent advantages in terms of storage requirements. An MT class can be implemented based on such structures as well, but this involves some changes in its interface. A detailed description and analysis of the data structures is given in a companion paper [7].

Class *extractor* implements the data structures that maintain the internal state of query algorithms, the DAG traversal and the user-level query operations. Class *extractor* has three subclasses, which implement the static, the dynamic and the local query algorithm, respectively. An extractor must be instantiated on an object of class *MT*. Several extractors, possibly of different types, may share the same MT without interfering.

The primitives for handling vertex and cell attributes are provided by class *attribute manager*. Attribute manager is a base class that contains operations to read/write a generic attribute and to set/get its value. Subclasses of *attribute manager* are written by the user to deal with specific types of attributes of the application. The MT class can be instantiated by providing one or more objects of (user-defined subclasses of) class *attribute manager*; the resulting MT will use the capabilities of the given attribute managers to deal with cell and vertex attributes.

An attribute manager contains internally an array of vertex attributes and an array of cell attributes; the size of the elements of such arrays can be decided by the user. When defining a subclass of *attribute manager* for handling a certain type of attribute, the size will be set accordingly. One of the array may not be used, if attributes are attached just to one of vertices and cells.

Module MT-tracer is implemented in class *tracer*. This class contains primitives that the user has to call for recording each vertex, cell, and update performed during a refinement or coarsening process. As the MT, the tracer can be instantiated by providing one or more objects of class *attribute manager*; in this case, the operations of the attribute managers permit to record a history that contains application-dependent attributes in addition to the standard basic information.

The MT-builder module is implemented in class *builder*; such class implements the basic and the user-level operations for constructing an MT either top-down or bottom-up from a history. As the tracer, it relies on attribute managers to build an MT containing user-defined attributes.

Module MT-query-params consists of class *query-param*, a base class whose subclasses are the user-defined focus sets and resolution filters. Class *query-param* can be instantiated on attribute managers in order to define filters which take application-dependent attributes of cells into account (e.g., error-based filters).

6.2. Software already implemented

The dimension independent library has been designed by extending structures and concepts that we have already developed and implemented in the two- and three-dimensional cases. The libraries at fixed dimension have the same architecture that we described in Section 3.1, but they are not parametric, neither on dimensions of space and complex nor on attributes of vertices and cells.

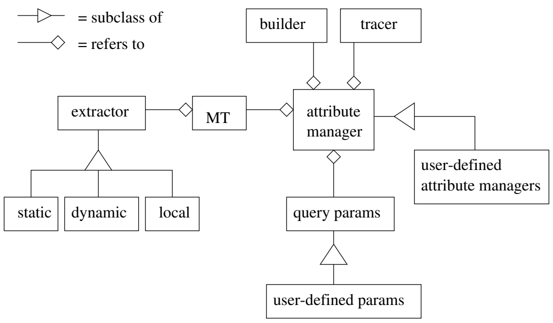


Figure 15: *The three levels of query operations.*

Both libraries have been implemented based on the data structure briefly outlined in Subsection 6.1. We are also developing alternative implementations that adopt more compact data structures, which are suitable for specific cases, such as Delaunay simplicial complexes, and two-dimensional MTs built through either vertex insertion or vertex decimation. Such alternative data structures are discussed and compared in [7].

Implementations have been made under the Unix operating system, and by now they have been tested on SGI (Irix) and on PC (Linux) platforms.

History generators. We have implemented different MT-history-generators: for two-dimensional fields (terrains) based on a Delaunay triangulation of a plane domain, we have developed different refinement and simplification methods based on iterative insertion and removal of vertices, respectively [5]; for free-form surfaces embedded in space, we have implemented the simplification method based on vertex removal proposed in [1], and we are completing the implementation of a simplification method based on edge collapse; for three-dimensional fields (volume data) based on Delaunay tetrahedrizations, we have implemented a refinement method based on vertex insertion, proposed in [2].

ROIs and resolution filters. Different methods for testing triangular and tetrahedra cells against regions of interest and resolution filters have been implemented.

The following ROIs only apply to the case of MTs representing fields, because they are defined on the domain of the field: rectangular boxes, circles, straight-line segments, polylines, points, and radial sectors. For the specific case of terrains we have also implemented vertical trapezoids hanging downward from a straight-line segment in the three-dimensional space used for visibility computation.

More general ROIs in three-dimensional space are used for fields, free-form surfaces, and volume data: cuboidal boxes, spheres, points, straight lines, and view frustums.

Resolution filters are available requiring a uniform error level, or based on an error threshold variable in space, or requiring a uniform size of triangles. Variable-error thresholds have been considered which increase with the distance from a viewpoint based on a linear, quadratic, or exponential law. Such thresholds are useful in surface visualization for generating a representation that locally adapts its level of detail to the distance from the viewpoint. Other resolution filters can be obtained by restricting these basic ones to a ROI.

7. Concluding Remarks

Approaches based on Level-of-Detail can improve the performance of many applications handling large geometric datasets, provided that developers can include LOD technology easily and with a reasonable efficiency. In order to achieve this goal, a flexible and general set of tools and methods it necessary, which must be obtained by undergoing a process of standardization.

We have presented an open library based on a fairly general model, the MT, which encompasses all current LOD models based on simplicial decompositions. The library can be used for modeling objects of any dimension k embedded in a space of any dimension d. Moreover, the library is generic on the set of attributes that can be attached to vertices and cells of objects modeled, thus supporting different modeling contexts such as free-form surfaces, scalar and vector fields, parametric surfaces, terrains, volumetric object representations, surface-on-surface, etc.

In its current version, our library provides data structures and methods to store and access an MT in primary memory. General construction methods are provided, which build an MT starting from the history file of a generic mesh simplification algorithm. Specific construction methods for scalar fields, free-form surfaces, and volume data are provided as well.

Important related issues, which are the subject of our current and future work, include the need to trade-off efficiency and space requirements, the management of secondary storage, the use of LOD in the context of client-server and network architectures, the management of application-specific data.

Acknowledgments

The support of National Science Foundation (NSF) Grant "The Grand Challenge" under contract BIR9318183 is gratefully acnowledged.

This work has been also partially supported by the BRITE European project BRPRCT96.0150 "VENICE - Virtual ENvironment interface by sensory integration for Inspection and manupulation Control in multifunctional underwater vehicles".

References

1. A. Ciampalini, P. Cignoni, C. Montani, and R. Scopigno. Multiresolution decimation based on global error. *The Visual Computer*, 13(5), 1997.
2. P. Cignoni, L. De Floriani, C. Montani, E. Puppo, and R. Scopigno. Multiresolution modeling and rendering of volume data based on simplicial complexes. In *Proceedings of 1994 Symposium on Volume Visualization*, pages 19–26. ACM Press, October 17-18 1994.
3. P. Cignoni, C. Montani, and R. Scopigno. A comparison of mesh simplification algorithms. *Computers And Graphics*, 22(1):37–54, 1998.
4. M. de Berg and K. Dobrindt. On levels of detail in terrains. In *Proceedings 11th ACM Symposium on Computational Geometry*, pages C26–C27, Vancouver (Canada), 1995. ACM Press.
5. L. De Floriani, P. Magillo, and E. Puppo. Building and traversing a surface at variable resolution. In *Proceedings IEEE Visualization 97*, pages 103–110, Phoenix, AZ (USA), October 1997.
6. 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*, Las Vegas, Nevada, 1997.
7. L. De Floriani, P. Magillo, and E. Puppo. Data structures for simplicial multi-complexes. Technical Report DISI-TR-98-18, Department of Computer and Information Science, University of Genova (Italy), 1998. (submitted for publication).
8. L. De Floriani, P. Magillo, and E. Puppo. Efficient implementation of multi-triangulations. In *Proceedings IEEE Visualization 98*, Research Triangle Park, NC (USA), October 1998.
9. L. De Floriani and E. Puppo. Hierarchical triangulation for multiresolution surface description. *ACM Transactions on Graphics*, 14(4):363–411, October 1995.
10. L. De Floriani, E. Puppo, and P. Magillo. A formal approach to multiresolution modeling. In R. Klein, W. Straßer, and R. Rau, editors, *Geometric Modeling: Theory and Practice*. Springer-Velrag, 1997.
11. A. Guéziec, G. Taubin, F. Lazarus, and W. Horn. Simplicial maps for progressive transmission of polygonal surfaces. In *Proceeding ACM VRML98*, pages 25–31, 1998.
12. H. Hoppe. View-dependent refinement of progressive meshes. In *ACM Computer Graphics Proc., Annual Conference Series, (SIGGRAPH '97)*, pages 189–198, 1997.
13. J. Lee. Analysis of visibility sites on topographic surfaces. *International Journal of Geographic Information Systems*, 5(4):413–425, 1991.

14. 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 *Comp. Graph. Proc., Annual Conf. Series (SIGGRAPH '96), ACM Press*, pages 109–118, New Orleans, LA, USA, Aug. 6-8 1996.

15. D. Luebke and C. Erikson. View-dependent simplification of arbitrary polygonal environments. In *ACM Computer Graphics Proc., Annual Conference Series, (SIGGRAPH '97)*, pages 199–207, 1997.

16. Paola Magillo. *Spatial Operations on Multiresolution Cell Complexes*. PhD thesis, Dept. of Computer and Information Sciences, U. of Genova, 1999.

17. E. Puppo. Variable resolution terrain surfaces. In *Proceedings Eight Canadian Conference on Computational Geometry*, pages 202–210, Ottawa, Canada, August 12-15 1996. Extended version to appear under title "Variable resolution triangulations" in *Computational Geometry Theory and Applications*.

18. 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.