Efficient Implementation of Multi-Triangulations

Leila De Floriani, Paola Magillo
Dipartimento di Informatica e Scienze dell'Informazione – Università di Genova *
Enrico Puppo
Istituto per la Matematica Applicata – Consiglio Nazionale delle Ricerche †

Abstract

The Multi-Triangulation (MT) is a general framework for managing the Level-of-Detail in large triangle meshes, which we have introduced in our previous work. In this paper, we describe an efficient implementation of an MT based on vertex decimation. We present general techniques for querying an MT, which are independent of a specific application, and which can be applied for solving problems, such as selective refinement, windowing, point location, and other spatial interference queries. We describe alternative data structures for encoding an MT, which achieve different trade-offs between space and performance. Experimental results are discussed.

CR Categories and Subject Descriptors: E.4 [Data]: Coding and information theory – data compaction and compression; I.3.3 [Computer graphics]: Picture/image generation – display algorithms; I.3.5 [Computer graphics]: Computational geometry and object modeling – curve, surface, solid and object representations – geometric algorithms. I.3.7 [Computer graphics]: Three-dimensional graphics and realism – virtual reality.

1 Introduction

Managing the Level-of-Detail (LOD) of large geometric meshes has a relevant impact on applications, like visualization, virtual reality, and geographic data processing. This topic has received increasingly more attention in the last few years, and a variety of models and methods have been proposed in the literature (see Section 1.2).

Especially important problems arise from the integrated study of operations that must be performed on LOD models, and of data structures to encode such models. There is a necessary trade-off between space requirements of a data structure and performance of algorithms that operate on it. An evaluation of requirements and performance is therefore mandatory to assess the impact of LOD models in practical applications.

In our previous work, we have introduced the *Multi-Triangulation (MT)* [8, 19], a general framework for modeling surface meshes at variable LODs, which encompasses all models based on decomposition proposed in the literature as special cases [20]. We have proposed methods for building an MT and for extracting space-continuous LOD representations from it.

In this paper, we describe an efficient implementation of an MT. In order to achieve this result, we elaborate on our previous work by addressing the following issues:

- a general approach for the solution of queries on a mesh at different LODs (such as selective refinement, windowing, point location, and other interference queries).
- Alternative data structures for encoding an MT which achieve different trade-offs between space and performance.
- Query methods implemented on top of such data structures in the context of an object-oriented library that provides a kernel to develop MT-based applications.

1.1 LOD Management

Let us assume we are designing an application that deals with a surface described by a large mesh, and needs to perform various operations. We allow the application to exploit LOD features by storing many different representations of a surface as an MT, and by providing an intermediate layer of software that handles the MT model. Such software can provide alternative representations of the whole surface, or portions of it. Meshes are extracted from an MT according to accuracy requirements, and to geometric constraints provided by the application (e.g., points, lines, windows,...). The software handling the MT will be called an *MT server*, while the application will be called an *MT client*.

In designing the MT server, we exploit features of the MT framework, which allow us not only to reduce the size of data handled by the client, but also to maximize the quality of the resulting meshes. This latter goal is achieved by tuning the loss of accuracy introduced by simplification according to the needs of the running application. In particular, the server can offer the features described below.

Selective refinement. A mesh, whose LOD is continuously variable through space, can be obtained in real time. The client provides a threshold function, i.e., a mathematical law specifying how the loss of accuracy/resolution of the mesh should be distributed through space. For instance: a LOD decreasing with the distance from the viewpoint (e.g., in a flight simulator); a LOD enhancing accuracy in the proximity of regions of interest. The server provides the mesh formed by the smallest number of triangles in the MT necessary to satisfy the threshold.

Locality. The mesh can be bounded to a local region of interest, specified by the client. For instance: the portion of surface inside the view frustum (for rendering); a window that is interactively dragged through space (for surface browsing); the surface portions lying inside an arbitrary region, along a line, or around a point (for surface analyzers, and design tools in GIS and CAD applications).

Dynamic update. The mesh can be changed dynamically as the client changes its threshold and the region of interest (e.g., frame-to-frame transition in the navigation in virtual worlds).

Progressive transmission. The simplified mesh, possibly clipped to a region of interest, can be transmitted by increasingly finer levels of detail. This allows the client to interrupt transfer before completion - e.g. because of space/time constraints - while getting a meaningful result, though at a lower LOD than desired. This also allows the server to organize mesh extraction and transmission in a pipeline.

*Via Dodecaneso, 35, 16146 Genova, ITALY, Email: {deflo,magillo}@disi.unige.it
†Via de Marini, 6 (Torre di Francia), 16149 Genova, ITALY, Email: puppo@ima.ge.cnr.it

Geometry compression. By adopting a suitable technique to build the MT, specific features can be exploited to design a compressed structure. It is possible to trade off storage and performance to different degrees, in order to obtain an optimal balance between the computing power and the memory requirements.

1.2 Related Work

Apart from early work on quadtrees, and similar decomposition hierarchies [21], most work on selective refinement has been done quite recently (see [20] for a survey). Most LOD data structures are built through local updates, which are popular methods for mesh simplification (see [12, 20] for surveys). Different data structures adopt different hierarchical organizations, such as pyramids [1, 6], trees of nested meshes [7, 10, 16], higher dimensional embeddings [5], directed acyclic graphs (DAGs) [3, 19, 11], linear sequences [13, 15], and vertex hierarchies [22, 14, 17, 18].

Algorithms for selective refinement have been proposed for all data structures cited above. They are based on a top-down visit of the hierarchy [3, 1, 10, 14, 17, 18, 19, 22], on a bottom-up traversal of the hierarchy [16], on a breadth-first traversal of the surface [5, 7], or on a linear scan with backtracking of a sequence of modifications [13, 15]. Algorithms proposed in [3, 1, 14, 19, 22] may achieve linear time in the size of their output: the results they produce have different characteristics, and the cost of data structures they require varies. The remaining algorithms have higher time complexity, and they cannot achieve real-time performances, unless heavy heuristics are adopted.

However, all such algorithms perform selective refinement by traversing the LOD model, and building a new mesh from scratch each time LOD conditions change. We are aware of only one proposal supporting the dynamic update of an existing mesh upon change of the conditions [11], but it is restricted to uniform LODs. Locality issues, such as those involved in point location and windowing have been addressed briefly only in [1]. We are not aware of any work on the evaluation of performance of algorithms accessing the structure versus storage requirements of the data structures for encoding a LOD model, while we believe this issue is crucial to the practical applicability of LOD models.

2 The Multi-Triangulation

The *Multi-Triangulation* (MT) is a multiresolution surface model that we introduced in [8, 19].

The intuitive idea is the following: consider a process that starts with a coarse mesh and progressively refines it by performing a sequence of local updates. A *local update* is an operation that replaces a group of triangles with another group of triangles covering the same area. In the context of this paper, each local update is caused by the insertion of a new vertex: a portion of the mesh influenced by the new vertex is retriangulated with a fan of triangles incident at the new vertex (see Figure 1). Some updates of the sequence are *independent* (because they affect distinct portions on the surface), while other updates *depend on each other* (because some triangles introduced by the first update are removed by the second one). For instance, in Figure 1, updates 4 and 5 are mutually independent, and they both depend on update 3.

An MT abstracts from the *totally ordered* sequence of updates, and encodes a *partial order* describing the *mutual dependencies* between updates as a directed acyclic graph (DAG).

A Multi-Triangulation is defined as a labeled DAG M, where nodes represent mesh updates, and arcs represent relations between updates. Arcs are labeled with sets of triangles:

- each node n represents a mesh update (an artificial update represents the creation of the initial coarse triangle mesh).
- an arc $a = (n_1, n_2)$ exists whenever the update represented by node n_2 depends on the update represented by node n_1, i.e., if a non-empty subset $T_{1,2}$ of the triangles introduced by n_1 are removed by n_2; arc a is labelled with set $T_{1,2}$.

Each triangle t, which is part of the final mesh (i.e., it is not replaced by any update), does not appear in the label of any arc. In order to have each triangle represented in exactly one arc, we introduce a dummy node, called the *drain*, and a set of dummy arcs incident into the drain, labeled with the unreplaced triangles. Figure 2 shows the MT corresponding to the updates of Figure 1 (dummy arcs are shown dashed).

Each triangle t in an MT is characterized by a *global approximation error*, corresponding to the distance between t and the surface patch it describes. Triangles of an MT can be combined to form global meshes representing the original surface at a variable LOD.

A *cut* of an MT is a set of arcs containing exactly one arc for each path from the root to the drain. The collection of all the nodes which can be reached from the root without traversing any arc of a cut (called the nodes *before* the cut) represents a set of mesh updates which is consistent with the partial order. The triangle mesh resulting from such updates is the same as the collection of the triangles labeling the arcs of the cut (see Figure 2).

Selectively refined triangle meshes can thus be obtained from cuts. The resolution of a mesh increases/decreases while sweeping a cut forward/backward through the DAG.

In this paper, we consider an MT built from a decimation technique that we presented in [9]. At each step of decimation, a maximal set of independent vertices of bounded degree is selected, each vertex in this set is removed, and the resulting hole is retriangulated. The fan of triangles incident at a decimated vertex corresponds to a new node of the MT. The mesh resulting at the end of the decimation process is T_0. This technique guarantees that each update is formed by a bounded number of triangles (thus each node has also a bounded number of arcs entering and leaving it), and that the depth of the resulting DAG (i.e., the length of the longest root-to-drain path) is logarithmic in the total number of triangles in the MT.

3 Querying a Multi-Triangulation

Although triangle meshes are used as representations of surfaces for several purposes, it is possible to abstract a few basic geometric operations which are independent of the specific application, and provide the general basis for implementing application-specific operations. Basic query operations include *windowing*, *point location*, *line and region interference*, and intersection of the mesh with a *view frustrum*.

A multiresolution model, such as the MT, contains a collection of alternative meshes representing the same surface at different LODs. In this case, geometric queries have an additional degree of freedom, i.e. selecting on which mesh the query must be performed. The target mesh is selected by providing some resolution criteria suitable for the purpose of the application: in this way, it is possible to perform *geometric queries at variable resolution*. In addition, the extraction of one of the meshes encoded in the multiresolution model is an interesting operation in itself, also known as *selective mesh refinement*.

Thus, two ortogonal issues are involved when querying an MT:

1. a specification of the (possibly variable) *LOD* of the mesh which will be queried;
2. a specification of the type of *geometric operation* (i.e., windowing, point location, etc.) defining the query; a special dummy operation returning the mesh itself provides a way for implementing selective refinement.

We consider a general query operation on an MT that addresses both issues. The query is specified through the following parameters:

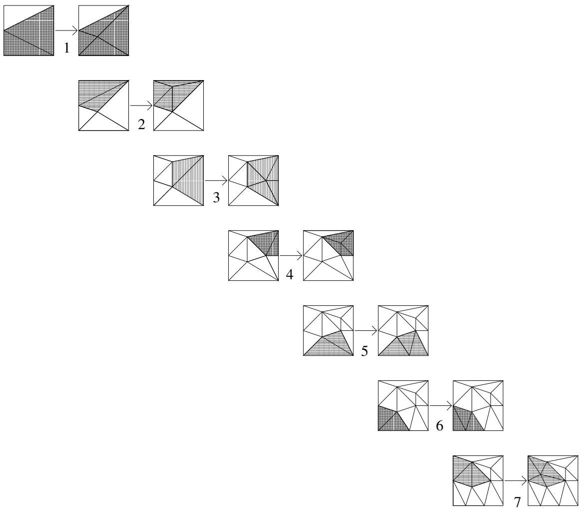

Figure 1: A sequence of local refinements of a triangle mesh.

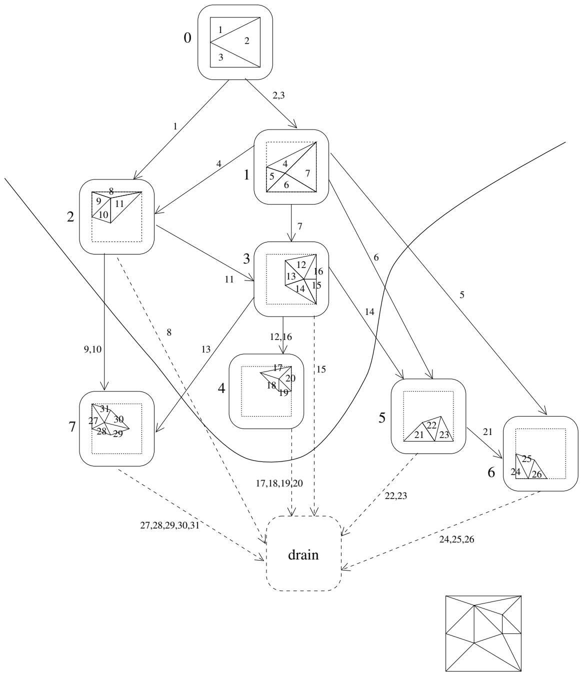

Figure 2: The DAG describing the MT corresponding to Figure 1, with a cut denoted by a thick curve; the mesh corresponding to the cut is shown on the right. For the sake of clarity, the triangles corresponding to the update described by a node are depicted in the node itself. The initial coarse mesh T_0 is shown in the root node.

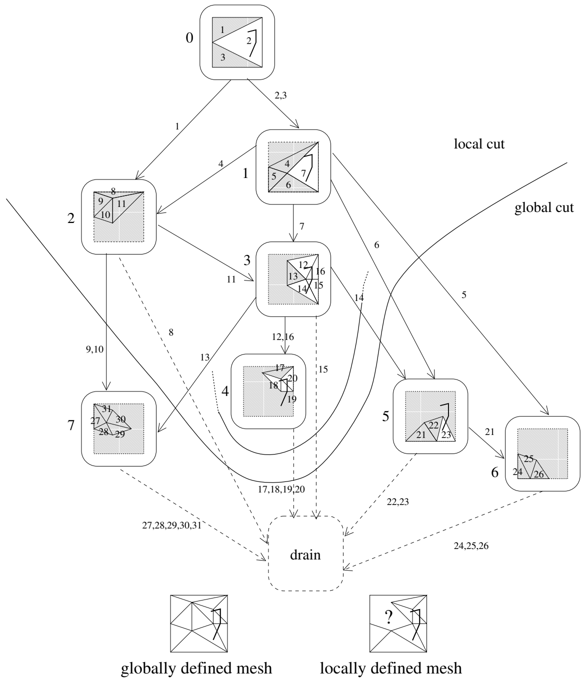

Figure 3: The globally and the locally defined meshes for a line interference query. Dashed (parts of) nodes are left out when computing the locally defined result.

- A *threshold function* $\tau : \mathbb{R}^3 \longrightarrow \mathbb{R}$ to bound surface approximation error at each point in space.
- A *focus set* $F \subseteq \mathbb{R}^3$ representing the region of interest of the query (e.g., a window, a point, a line, a region etc.).

We call a triangle t *valid* if and only if its approximation error is smaller than the minimum of the threshold function τ over t; we call it *active* if and only if it intersects the focus set F.

The answer to the query is the *smallest* mesh (i.e., the one with the fewest number of triangles), among those available in the MT, where all active triangles are valid.

We define two alternative instances of the general query, depending on additional properties desired for the resulting mesh:

- the mesh must be *globally defined*, i.e., the smallest mesh representing the whole surface, and approximating the area of interest within the given error bounds.
- the mesh can be just *locally defined*, i.e., the smallest mesh which covers the area of interest, and has an approximation error within the given error bounds.

A globally defined mesh extends consistently beyond the region of interest, although it does not guarantee any error bound outside it. A locally defined mesh cannot be always extended beyond the focus set by other triangles of the MT (see Figure 3). Globality is important, for instance, in animated visualization to avoid popping effects due to abrupt changes of the mesh when the view changes. Locally defined meshes can be used to evaluate local properties of the surface.

The globally defined result of the query is given simply by a suitable cut of the MT. The result of a local query can be regarded as a cut in a *clipped DAG*, obtained from the DAG describing the MT by deleting all non-active triangles, and all arcs labeled by non-active triangles only.

The two instances of the query with a globally and a locally defined output mesh will be called the *global* and the *local* query, respectively, for short.

The algorithms for solving the global and local queries on an MT combine a DAG traversal with a local interference test: the DAG traversal addresses the LOD issue and is parametric over the given threshold function; the interference test addresses the geometric issue and is parametric over the given focus set.

Algorithm for the global query. In [9] we described a DAG traversal strategy for the global query, which we refer here as algorithm **A**. The method works top-down, starting form the root of the MT, and moves the cut forward, until all active triangles in the associated mesh are valid.

In certain applications, several successive queries are applied to a mesh, where the result of the next query is only slightly different from the result of the previous one. For instance, in a virtual reality environment the focus set coincides with the view frustrum of a moving viewpoint, and the threshold function is increasing with distance from the viewpoint itself. The main drawback of algorithm **A** is that, in case of repeated queries, the process restarts from the top of the DAG, even if the resulting cut for the new query is very close to the previous one.

Here, we present an algorithm, called algorithm **B**, which updates a previously computed cut to adapt to changes in the query parameters. The algorithm sweeps the cut forward and backward by moving it either from before a node to after that node (refinement), or vice-versa (simplification). A cut can be moved after [before] a node if and only if it contains all the arcs entering [leaving] that node. The nodes having all their outgoing arcs in the current cut are called *front nodes*. The current cut is initialized with the input cut, then the algorithm performs the following two steps:

1. *Refinement*: the current cut is scanned, and a local refinement is requested each time an active and non-valid triangle t is found. If the end node n of the arc containing t has all its incoming arcs in the cut, then the cut is simply moved after n; otherwise, a request for a local refinement is recursively propagated to all ancestors of n that lie before the cut.
2. *Simplification*: the front nodes are scanned, and each time all active triangles labeling the incoming arcs of a node n are valid, the cut is moved before n.

A running example is shown in Figure 4.

Algorithm **A** can be seen as a special case of algorithm **B**, which always starts at the cut just after the root node, and performs only the refinement step.

Let A, and V be the costs of evaluating whether or not a triangle is active, and valid, respectively, and let k be the size of the output cut. The time complexity of the algorithm is $O((A+V)n_k)$, where n_k is the number of triangles tested. n_k is the sum of the number of triangles labeling the arcs swept by the algorithm, and the arcs of the input and output cuts; it is linear in the sum of sizes of the input and output triangle meshes. Note that algorithm **A** has a computational complexity of $O((A + V)k)$. In particular, if the threshold and focus conditions can be evaluated in constant time at each triangle, the complexity is $O(k)$, hence optimally output-sensitive.

Another dynamic algorithm was proposed by Guéziec et al. [11]. Also this algorithm is based on adjusting a cut on a DAG, but it uses a priority queue to decide which nodes the cut can be swept after or before. The priority in the queue depends on the approximation error of the old and of the new mesh of a node. However, the approach is restricted to uniform error thresholds and cannot deal with focus sets.

Algorithm for the local query. For the local query, we propose a new algorithm, called algorithm **C**, which adapts algorithm **A** to deal with partial cuts. It is essentially a combination of algorithm **A** with a culling mechanism for eliminating non-active triangles.

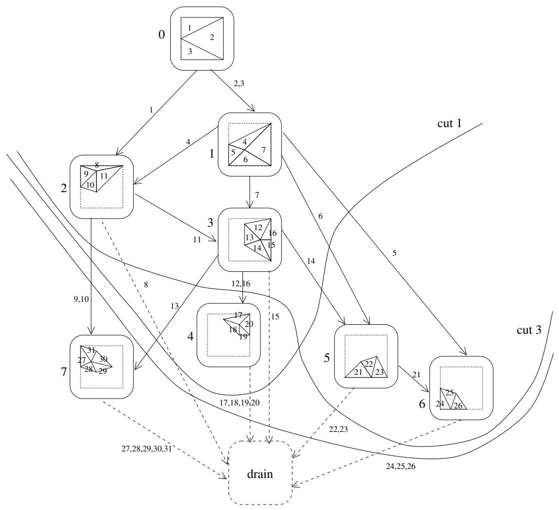

Figure 4: Algorithm **B** at work. Cut 1 is the input cut. Triangle 5 is active and not valid, triangles 12, 16 are not active. Refinement stage: the cut is advanced after node 6; since node 5 is before the cut, node 5 is swept as well; The resulting cut is cut 2. Simplification step: the cut is swept before node 4; cut 3 is the ouput cut.

As algorithm **A**, algorithm **C** starts from the cut just after the root, and progressively advances the cut. The difference is that all arcs labeled only by non-active triangles are purged out of the current cut. For convenience, we call *active* an arc having at least one active triangle in its label.

The current cut is initialized with the set of active arcs outgoing from the root. Then, the current cut is scanned, and a local refinement is requested each time an active and non-valid triangle t is found. Let n be the end node of the arc containing t. If all active arcs entering n are in the cut, then the cut is simply moved after n; otherwise, a request for a local refinement is recursively propagated to all ancestors of n that lie before the cut, and are connected to n through active arcs.

The culling mechanism allows the algorithm to purge rapidly those portions of the MT that lie outside the focus set. The time complexity is $O((A + V)m_k)$, where m_k is the number of triangles tested. m_k is the number of triangles labeling the arcs lying on paths from the root to the final partial cut, including the ones of the final partial cut; it is linear in the size k of the output locally defined triangle mesh. In particular, if the threshold and focus conditions can be evaluated in constant time at each triangle, the time complexity is $O(k)$, hence optimally output-sensitive.

Note that a dynamic approach like the one used in algorithm **B** does not make sense for the local query: the partial cut does not give enough support for allowing its interactive modification when the query paramaters (in particular, the focus set) change.

4 Data Structures for an MT

In order to support the algorithms described in Section 3, we must either encode information explicitly, or provide suitable mechanisms to retrieve information on-line.

The space complexity of an MT is characterized by: the number of nodes `n`, the number of arcs `a` in the DAG describing it, the total number of triangles `t`, and the total number of vertices `v`. Our experimental results give the following empirical evaluations for triangles, arcs, and nodes in an MT: $\mathtt{t} \approx 5\mathtt{v}$, $\mathtt{a} \approx 3\mathtt{v}$, $\mathtt{n} \approx \mathtt{v}$. In the following analysis, we will rely on such estimates in order to compute space costs as a function of the number of vertices.

We describe two different ways of storing an MT, and we compare their costs and performances. In order to estimate the overhead of our data structures, we also compare them to a raw indexed data structure encoding the mesh at the highest resolution. This consists of an array of vertices, each represented by a three floats, plus an array of triangles, each represented by three indexes to vertices in the first array. Since the number of triangles in a mesh is nearly twice the number of its vertices, the indexed data structure contains $3\mathtt{v}$ floats and $6\mathtt{v}$ indexes. We assume to work with words of 32 bits, encoding each float with a word, and packing three indexes in two words. This allows us to use 21 bits per index, addressing up to 2 millions vertices, which is sufficient for most surface meshes. Therefore, the cost of the indexed data structure is $7\mathtt{v}$ words.

The explicit data structure. The direct implementation of the MT is called the *explicit data structure*. It stores four arrays containing vertices, triangles, arcs, and nodes of an MT, respectively:

- Each entry in the vertex array stores the three cartesian coordinates of a vertex v.
- Each entry in the triangle array entry stores: the indexes of the three vertices of a triangle t; its approximation error; and the index of the arc having t in its label. Triangles labeling the same arc are stored at consecutive entries.
- Each entry in the arc array entry stores: the indexes of the source and destination nodes of an arc a; the index of the first triangle belonging to the label of a; the number of triangles in the label of a; and the index of the next arc having the same destination node. Arcs having the same source node are stored at consecutive positions.
- Each entry in the node array entry stores: the index of the first arc outgoing from a node n; the number of arcs outgoing from n; the index of the first arc entering n. The root and the drain nodes are stored in the first, and the last position, respectively.

By packing information as before, we need one word per float, 20 bits per index, and 16 bits for the (quantized) error. Four bits would be sufficient to count triangles in a label, and arcs incident at a node, since we always bound the number of triangles in each update to a number smaller than sixteen. However, we may use up to 16 bits for each such number in order to round-off words. With such assumptions, the total cost of the explicit data structure is $3\mathtt{v} + 3\mathtt{t} + 3\mathtt{a} + 2\mathtt{n} \approx 29\mathtt{v}$ words, hence nearly four times the cost of the indexed data structure. This gives an overhead of about 300% with respect to the raw indexed structure. Note that a large part of the overhead is due to maintaining triangles explicitly.

The size of the explicit data structure is comparable to those adopted in [3, 1, 14, 19], in terms of both information encoded, and storage cost.

The implicit data structure. The *implicit data structure* encodes the DAG, but not the triangles. The local mesh corresponding to each node is obtained on-line by exploiting the special characteristics of the simplification method used to build the MT (i.e., each node is a fan of triangles).

The structure consists of three arrays, representing vertices, nodes, and arcs of the DAG, respectively:

- The vertex array contains the same information as the one used in the explicit data structure.
- Each entry in the arc array stores the indexes of the source and destination nodes of an arc, and the index of the next arc having the same destination node.
- Each entry in the node array stores the same information as the explicit structure, plus the maximum approximation error of the triangles forming the update associated with the node, and *local information* to define such an update (as described below).

The initial mesh needs a special structure: its triangles can be maintained either explicitly, or through some compressed structure; since it is always a small mesh, we disregard its cost. In this assumption, the cost of maintaining the vertex and arc arrays is $9\mathtt{v}$ words. The cost of the node array depends on local information.

We give two alternatives for local information: one that can be used in general, and a more compressed one that can be used for bivariate surfaces based on Delaunay triangulation.

A local update, corresponding to a node n, is performed by inserting a vertex p, and retriangulating its *influence area*. First, point p splits a triangle into three; then, a sequence of edge swaps is performed. (see Figure 5). Thus, each update is fully described by the new vertex p, a reference triangle t, and a list of edge swaps.

The reference triangle t always corresponds to a parent m of node n. We can always put m at the source of the arc pointed by n. Therefore, a reference to m can be obtained at no extra cost. If we order the triangles associated with each node conventionally, then triangle t is identified by an index in m. Splitting t produces a fan of three triangles incident at p. Each edge swap affects one triangle in the current fan, and increases the number of its triangles by one unit. We encode the swap through the index of such a triangle in the sorted sequence of triangles incident at p. Since the number of triangles per node is bounded by a constant b, then the number of swaps is bounded by $b - 3$. The index of t within m is in the range $0 \dots b - 1$ and requires $\log_2 b$ bits. The first swap index is in $0..2$, and the i-th swap is in the range $0..(i + 2)$. Therefore, the sequence of swaps can be packed in a *swap code* of $\sum_{i=3}^{b-1}(\log_2(i)) = \log_2((b-1)!) - 1$ bits. We can pack the swap code and the index of t together in $\log_2(b!) - 1$ bits.

We have empirically found that a value $b = 11$ gives the best results in the quality of the MT. In this case, $4\mathtt{v}$ words are sufficient to store all node information. In conclusion, the cost of the implicit data structure is $13\mathtt{v}$ words, with only a 85% overhead with respect to the raw indexed structure.

For surfaces that can be projected on a plane, such as terrains, it is possible to update meshes on the basis of the Delaunay criterion. A triangulation is called a Delaunay triangulation if the circumcircle of any of its triangles does not contain vertices in its interior. In this case, the sequence of swaps caused by the insertion of a vertex p can be computed on-line on the basis of suitable geometric tests [2]. Therefore, we just need to encode the index of p, hence saving one memory word, and reducing overhead to 70%. The reduced storage cost is paid in terms of numerical computations during mesh extraction (see Section 5).

5 Implementation

We have developed an object-oriented library implementing the Multi-Triangulation and related algorithms, thus providing a kernel for MT-based applications. The library is written in C++ and fully portable. By now, it has been tested on SGI and PC platforms. It implements the traversal algorithms described in Section 3, and provides a collection of focus sets and threshold functions which can be used as parameters for the algorithms. User-defined focus sets and threshold functions can be simply defined by adding new classes to the library. Focus sets and threshold conditions suitable for both bivariate surfaces (terrains) and free-form surfaces are provided. In the case of bivariate surfaces a focus set is defined on the xy plane, and the query algorithms tests active triangles on the projection of the mesh on the xy plane. This makes operations such as

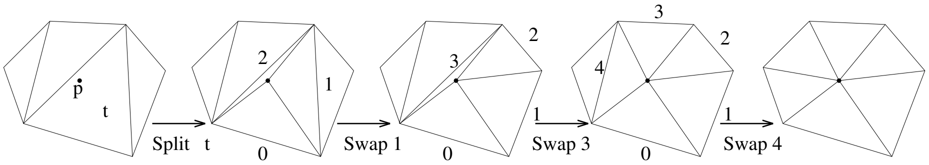

Figure 5: Insertion of a new vertex p by performing a triangle split and a sequence of edge swaps.

point location, and 2D windowing very easy.

For free-form surfaces, detecting active triangles is more difficult because approximation can warp the surface, bringing a triangle out of the focus set, though its corresponding portion of surface lies inside it. We adopted a conservative technique to detect active faces. A triangle t with an error ε is considered active when the focus set intersects the wedge obtained by sweeping the triangle along its normal direction, for a distance ε above and below it (the portion of surface approximated by such a triangle is guaranteed to lie inside such a wedge). This allows the algorithm to perform easily queries such as ray shooting, and 3D windowing.

All algorithms described in Section 3 have been implemented for the explicit data structure, while only algorithm **A** has been implemented for the implicit data structure in the variant for Delaunay-based meshes. The general version of the implicit data structure is currently under implementation.

Implementation follows directly from the specification when using the explicit data structure. Besides a primary structure encoding the MT, the algorithms maintain locally the following dynamic structures: a list of arcs in the current cut, and a bit vector used to mark nodes lying before the current cut. Algorithm **B** maintains also a list of all front nodes, and a list of nodes having arcs in the cut, storing the number of arcs in the cut for each node. The algorithms simply locate the cut corresponding to the output mesh, and report all triangles labeling such a cut.

Implementation with the implicit structure presents two problems:

1. Since triangles are not stored, the result of each local update must be computed from the current mesh, plus local information at the node corresponding to the update operation.
2. The error stored in each node provides only an upper bound on the approximation error of the triangles involved in the corresponding update, and we have no information about how such triangles are distributed among the arcs of the node.

Problem 1 involves additional work to reconstruct the current mesh at each step. Moreover, the refinement algorithm needs to maintain a local data structure describing the mesh corresponding to the current cut, including triangle adjacencies, in order to support mesh updates. By now, we have implemented only algorithm **A** for the global query, only for an MT representing bivariate functions (terrains) based on the Delaunay triangulation of the vertices projected on the xy plane. In this case, refinement is based on a well-known algorithm for updating a Delaunay triangulation through online insertion of vertices [2]. Such an algorithm requires locating the new vertex inside a triangle of the current mesh, followed by a sequence of in-circle tests to find triangles that are influenced by the new vertex. This involves floating point operations which obviously affect practical performance considerably. On the contrary, performing a mesh update with the general implicit structure is easier, since it reduces to applying a pre-computed sequence of edge swaps, with no numerical computations involved. For this reason, we expect that performance can be highly improved with such a version of the implicit structure.

Problem 2 is intrinsic of any structure that does not store triangles explicitly (e.g., also to data structures in [14, 15, 17, 18, 22]), and affects the size of the output mesh. The error stored in each node provides a pessimistic estimate to test whether a triangle is valid or not. Moreover, when an active triangle t is not valid, the algorithm does not know which child of node n that created t gives the update to remove t. Therefore, either all children of n are expanded, and tested to find the desired child (with a further increase of computational costs), or the current cut is swept after all children of n. We implemented the second, simpler, alternative. For the above reasons, the output mesh may be over-refined considerably with respect to the needs of the threshold function (see Section 6).

	Explicit structure		Implicit structure	
error	time	triangles	time	triangles
20%	0.68	176	69.3	566
5%	5.41	1,281	319.5	3,726
1%	39.05	8,438	1,543.9	19,756
0.2%	252.05	37,073	3,554.5	51,900
0.0%	462.32	69,718	4,820.6	69,718

Table 1: Results of refinement at uniform resolution for the explicit and for the implicit data structures. Times are in milliseconds.

6 Experimental Results

Experiments have been performed on an SGI Indigo2 workstation (R10000, 195 Mhz, 128 Mb RAM), on several datasets. Here, we present experiments on two MTs that are representative of the results we obtained on all real world examples.

- A terrain MT, built with a refinement technique described in [9], from an input grid of 200 × 200 points representing a terrain in the area of Mount Marcy, NY, USA (courtesy U.S. Geological Survey).
- A free-form MT built by a decimation technique described in [4], from an input mesh with 69,451 triangular faces, representing a bunny (courtesy Department of Computer Science, Stanford University).

Comparison of explicit and implicit data structures. We have compared the explicit and implicit data structure only on the terrain MT. The MT for Mount Marcy contains 35,160 nodes, 103,093 arcs, 175,922 triangles and 35,162 vertices. The sizes of the explicit and of the implicit data structure are about 6.5 and 2.5Mb, respectively. The space of the explicit structure is about three times that of implicit one. Similar results have been obtained on other terrain MTs. Table 1 compares the extraction time and the size of the extracted triangulation in the explicit and implicit case, respectively. We used the whole domain as the focus set, and a constant threshold function, whose value is expressed as a percentage of the range of elevation spanned by the dataset.

With the implicit data structure, extraction times increase of an average factor of ten, and the sizes of the resulting meshes are about

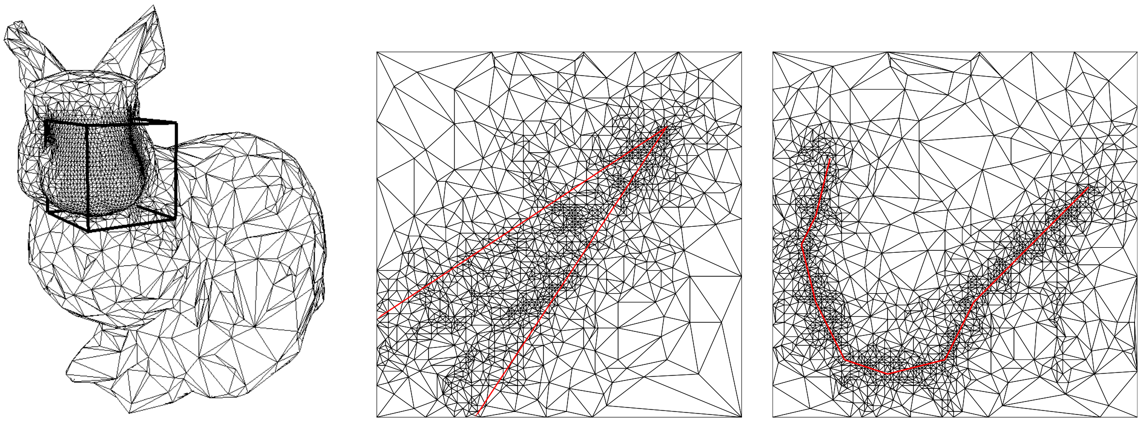

Figure 6: Extracted meshes from the MTs representing bunny, and Mount Marcy.

doubled, with respect to the results obtained with the explicit structure. The increase in the sizes of the meshes is due to the impossibility of testing the accuracy of each triangle individually, as discussed in Section 5. The increase in extraction times is due to the on-line computation of a Delaunay triangulation, and to the fact that a larger portion of the DAG is traversed.

Comparison of the Three Query Algorithms. Algorithms **A**, **B** and **C** have been implemented with the explicit data structure, and compared on both the terrain and the bunny MT. For the bunny, we have considered an error threshold equal to zero, and a focus set consisting of an axis-parallel box which moves along a trajectory. For Mount Marcy, the threshold function is linearly increasing with distance from a given viewpoint, which moves along a trajectory, and the focus set is a sector centered at the viewpoint, and having its axis along the trajectory. Examples of extracted meshes are shown in Figure 6.

Table 2 reports the number of triangles in the output mesh, the number of triangles swept by the cut, the number of triangles tested against the extraction conditions, and execution times, for the three algorithms. The same information is shown in graphics in the color plate. For the global query, the extracted mesh describes the whole surface, while for the local query it extends only within the focus area. In the experiments, the size of the mesh extracted by the local query is almost identical to the number of active triangles in the mesh extracted by the global one.

In the all approaches, the number of tested triangles is roughly proportional to the output size. The number of triangles swept by algorithms **A** and **C** is proportional to the size of the output mesh, while for algorithm **B** it is very low when the mesh changes gradually.

Local extraction time is much lower for the bunny, because the focus set used on the terrain is less selective. In any case, it is generally proportional to the number of active and valid triangles reported.

7 Concluding Remarks

LOD models may be useful in more contexts than rendering applications. For this reason, it is necessary to study techniques that combine traversals at differents LODs with efficient mechanisms to perform spatial queries. We have shown that in our implementation performance is achieved in supporting a number of common queries of interest in different applications. The good behaviour of our query operations is largerly due to the construction algorithm that produces an MT with logarithmic depth, and with a bounded number of triangles per update.

The compared analysis of performance and storage cost of MT models is a crucial issue for their practical use. We have shown that our decimation-based MT encoded with the explicit structure may achieve real-time performance in processing large meshes, while its storage requirements are comparable with those of other efficient structures proposed in the literature. However, since the overhead introduced by such data structures may be relevant, processing huge meshes by maintaining one such LOD model in primary storage may be prohibitive for mid-level workstations. It is an open issue whether this problem should be solved by addressing efficient mechanisms for handling data in secondary storage, or by adopting more compact data structures with a lower overhead. The implicit structure for Delaunay meshes highly reduces storage cost, but its performances suffer from numerical computation involved during traversal. The more general implicit structure based on swap codes seems more promising to achieve a good balance between storage and performance.

It should be noted that similar results can be obtained also on other variants of the MT. For instance, an MT whose local updates are based on vertex split operations gives an instance of Progressive Meshes [13]. If such an MT is produced by iteratively collapsing independent sets of edges (as in [22]), whose end vertices have bounded degree, then logarithmic depth and a bounded number of triangles per update can also be achieved. In this case, on the basis of arguments similar to those used in Section 4, MT nodes can be encoded in a compressed form.

Acknowledgement: This work has been partially supported by the project "A Library for Applications in Geometric Modeling" of the Italian National Research Council.

References

[1] M. de Berg, K.T.G. Dobrindt, On the Levels of Detail in Terrains, *Proceedings 11th ACM Symposium on Computational Geometry*, Vancouver, pp.c26-c27, 1995.

[2] J.D. Boissonnat, M. Teillaud, On the Randomized Construction of the Delaunay Tree, *Theoretical Computer Science*, 112, pp.339-354, 1993.

[3] P.J.C. Brown, A Fast Algorithm for Selective Refinement of Terrain Meshes, *Proceedings COMPUGRAPHICS 96*, pp.70-82, 1996.

[4] A. Ciampalini, P. Cignoni, C. Montani, R. Scopigno, Multiresolution Decimation Based on Global Error, *The Visual Computer*, 13, 5, pp.228-246, 1997.

	extracted triangles		swept triangles			tested triangles			extraction time (msecs)		
iter.	**global**	**local**	**A**	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**
5	2681	1879	11663	326	8823	3922	3195	7410	53.77	36.07	54.02
15	2734	1845	11900	38	8735	3952	3091	7273	56.51	36.03	55.31
25	2736	1811	11902	260	8555	3931	3217	7138	55.24	35.68	54.88
35	2758	1807	11996	44	8575	3960	3123	7106	55.31	35.48	55.53
45	2792	1798	12114	134	8550	4046	3232	7101	54.47	35.38	54.75
55	2640	1752	11450	14	8428	3806	2992	6968	58.17	36.25	54.99
65	2688	1685	11688	242	8113	3871	3133	6795	56.28	36.37	53.19
75	2768	1713	12014	266	8267	3986	3163	6901	55.69	35.43	51.65
85	2710	1639	11770	207	7663	3870	3157	6384	53.68	34.85	51.83
95	2682	1627	11644	90	7743	3844	3023	6458	54.04	34.69	51.84

(Mt. Marcy)

	extracted triangles		swept triangles			tested triangles			extraction time (msecs)		
iter.	**global**	**local**	**A**	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**
5	2230	78	4166	62	656	2538	2252	2287	39.31	25.76	14.10
15	2426	222	5094	158	1224	2837	2476	2684	42.52	27.77	16.99
25	2572	354	5776	0	2064	2982	2628	3223	43.12	28.29	20.41
35	2690	426	6354	50	2320	3214	2760	3408	44.58	28.90	21.43
45	2749	443	6713	268	2415	3290	2973	3424	45.33	30.25	21.42
55	2745	446	6667	134	2444	3253	2824	3387	44.85	29.24	21.03
65	2602	440	5858	624	2416	2956	3031	3368	41.36	29.20	20.87
75	2688	450	6312	56	2346	3163	2762	3348	43.34	28.20	20.70
85	2690	458	6372	20	2336	3165	2764	3322	43.50	28.20	20.61
95	2584	413	5836	270	2151	2939	2801	3218	40.94	27.94	19.74

(Bunny)

Table 2: Experimental results for the three extraction algorithms. See also the color plate.

[5] P. Cignoni, E. Puppo, R. Scopigno, Representation and Visualization of Terrain Surfaces at Variable Resolution, *The Visual Computer*, 13, 5, pp.199-217, 1997.

[6] L. De Floriani, A Pyramidal Data Structure for Triangle-Based Surface Description, *IEEE Computer Graphics and Applications*, 8, pp.67-78, 1989.

[7] L. De Floriani, E. Puppo, Hierarchical Triangulation for Multiresolution Surface Description, *ACM Transactions on Graphics*, 14, 4, pp.363-411, 1995.

[8] L. De Floriani, E. Puppo, P. Magillo, A Formal Approach to Multiresolution Hypersurface Modeling, *Geometric Modelin: Theory and Practice*, W. Straßer, R. Klein, R. Rau (Editors), Springer-Verlag, 1997.

[9] L. De Floriani, P. Magillo, E. Puppo, Building and Traversing a Surface at Variable Resolution, *Proceedings IEEE Visualization'97*, pp.103-110, 1997.

[10] M. Duchaineau, M. Wolinsky, D.E. Sigeti, M.C. Miller, C. Aldrich, M.B. Mineed-Weinstein, ROAMing Terrain: Real-time Optimally Adapting Meshes, *Proceedings IEEE Visualization'97*, pp.81-88, 1997.

[11] A. Guéziec, G. Taubin, F. Lazarus, W. Horn, Simplicial Maps for Progressive Transmission of Polygonal Surfaces, *Proceeding ACM VRML98*, pp.25-31, 1998.

[12] P. Heckbert, M. Garland, Survey of Polygonal Surface Simplification Algorithms, *Siggraph'97 Course Notes*, 1997.

[13] H. Hoppe, Progressive Meshes, *SIGGRAPH 96 Conference Proceedings*, pp.99-108, 1996.

[14] H. Hoppe, View-Dependent Refinement of Progressive Meshes, *SIGGRAPH 97 Conference Proceedings*, 1997.

[15] R. Klein, W. Straßer, Generation of Multiresolution Models from CAD Data for Real Time Rendering, *Theory and Practice of Geometric Modeling*, W. Straßer, R. Klein, R. Rau (Editors), Springer-Verlag, 1996.

[16] P. Lindstrom, D. Koller, W. Ribarsky, L.F. Hodges, N. Faust, G.A. Turner, 1996, Real-Time, Continuous Level of Detail Rendering of Height Fields, *SIGGRAPH 96 Conference Proceedings*, pp.109-118, 1996.

[17] D. Luebke, C. Erikson, View-Dependent Simplification of Arbitrary Polygonal Environments, *SIGGRAPH 97 Conference Proceedings*, pp.199-208, 1997.

[18] A. Maheshwari, P. Morin, J.-R. Sack, Progressive TINs: Algorithms and Applications, *Proceedings 5th ACM Workshop on Advances in Geographic Information Systems*, Las Vegas, 1997.

[19] E. Puppo, Variable Resolution Terrain Surfaces, *Proceedings Eight Canadian Conference on Computational Geometry*, pp.202-210, 1996. Also available in extended version as: Variable resolution triangulations, *Technical Report 16/96*, IMA-CNR (submitted for publication).

[20] E. Puppo, R. Scopigno, Simplification, LOD, and Multiresolution - Principles and Applications, *Eurographics'97 Tutorial Notes*, 1997.

[21] H. Samet, *Applications of Spatial Data Structures*, Addison-Wesley, Reading, MA, 1990.

[22] J.C. Xia, J. El-Sana, A. Varshney, Adaptive Real-Time Level-of-Detail-Based Rendering for Polygonal Models, *IEEE Transactions on Visualization and Computer Graphics*, 3, 2, pp.171-183, 1997.