A Multi-Resolution Topological Representation for Non-Manifold Meshes

Leila De Floriani, Paola Magillo, Enrico Puppo, Davide Sobrero
Dipartimento di Informatica e Scienze dell'Informazione
Università di Genova
Via Dodecaneso, 35, 16146 Genova, ITALY
{deflo,magillo,puppo,sobrero}@disi.unige.it

Corresponding author: Paola Magillo

Abstract

We address the problem of representing and processing 3D objects, described through simplicial meshes, which consist of parts of mixed dimensions, and with a non-manifold topology, at different levels of detail. First, we describe a multi-resolution model, that we call a Non-manifold Multi-Tessellation (NMT), and we consider the selective refinement query, which is at the heart of several analysis operations on multi-resolution meshes. Next, we focus on a specific instance of a NMT, generated by simplifying simplicial meshes based on vertex-pair contraction, and we describe a compact data structure for encoding such a model. We also propose a new data structure for two-dimensional simplicial meshes, capable of representing both connectivity and adjacency information with a small memory overhead, which is used to describe the mesh extracted from an NMT through selective refinement. Finally, we present algorithms to efficiently perform updates on such a data structure.

keywords: Non-manifold modeling, multi-resolution, data structures.

1 Introduction

The problem of representing and manipulating spatial objects with a complex topology and formed by parts of different dimensionalities has been faced in solid modeling long time ago (see, e.g., [20, 33, 36]) because of its relevance in CAD/CAM applications. Non-manifold modelers allow combinations of wire-frame, surface, solid and cellular decompositions of a 3D object to be represented and manipulated in a consistent topological representation. Complex non-manifold objects are described through meshes with a non-manifold and non-regular (i.e., with parts of different dimensionalities) domain, encoded in non-manifold data structures. Motivations for using non-manifold data structures have been pointed out by several authors [2, 20, 33, 36]. For instance, Boolean operators are closed in the non-manifold domain, sweeping or offset operations may generate parts of different dimensionalities, non-manifold topologies are required in different product development phases, such as conceptual design, analysis or manufacturing [2, 34].

Modern acquisition devices and Web technology have made very large non-manifold meshes widely available. Because of their size and complexity, such meshes are often hard to handle by visualization, analysis or by algorithms for performing standard modeling operations. Thus, the need for producing smaller-size, simplified meshes arises for both reducing space requirements and enhancing computational performances. Moreover, for many operations, a detailed representation of the whole object may not be necessary, while a detailed description only in some significant part of the object may be sufficient. Such a description is obtained by a mesh whose level of detail is variable in space according to the application needs. Such meshes, usually called *selectively refined* meshes, can be obtained by a process of on-line simplification which removes details from a large dense mesh.

Mesh simplification algorithms have been developed in the computer graphics literature for visualization purposes (see [3, 16, 26]), most of which working on manifold triangle meshes. Most of the algorithms which deal with non-manifold topologies are based on are based on vertex-pair contraction (see, for instance, [13, 17, 30]). On the other hand, accurate algorithms can be too time consuming to be performed on-line, and also they do not generate a topological data structure for describing the resulting non-manifold mesh.

Therefore, it is convenient to de-couple the simplification phase, which is performed off-line, from the selective refinement phase, which is performed on-line. A multi-resolution model is built in a preprocessing phase, which encodes the modifications performed by the simplification algorithm as a partial order. A virtually continuous set of meshes at different Levels-Of-Detail (LODs) can be interactively and dynamically obtained from such model. Such approach has been extensively applied for view-dependent rendering of triangle meshes [13, 19, 22, 27, 37].

In our previous work, we have developed a completely general multi-resolution model for triangle meshes with a manifold domain [31], called a *Multi-Tessellation* (MT). We have defined and implemented instances of such model which are dependent on the specific simplification operator applied in its construction (see De Floriani and Magillo [5] for a recent survey). Our approach is not restricted to rendering, since we want to allow navigation and geometric operations at different LODs.

Here, we address the problem of representing and processing non-regular, non-manifold two-dimensional simplicial meshes, that we call *triangle-segment* meshes, at different Levels-Of-Detail (LODs). The MT extends naturally to the non-manifold domain as a *Non-manifold Multi-Tessellation* (NMT), which is a general model for representing such structures at variable resolution. However, the extension of algorithms and data structures developed for the MT (in the manifold case) to the non-manifold case is not straightforward.

We focus on a specific instance of a NMT built on the basis of the most common operation for non-manifold mesh simplification, i.e., vertex-pair contraction. We consider the selective refinement query, since this is at the heart of analysis operations on multi-resolution meshes. Selective refinement consists of extracting a mesh of minimal size having a resolution variable in different parts of the object according to user-defined requirements. Unlike existing proposals for view-dependent rendering of non-manifold meshes [13, 27], we are interested in extracting meshes with a complete topological description to support mesh traversal through adjacencies, as required by geometric modeling operations.

Major contributions of this paper are:

- A compact data structure for a specific instance of a NMT based on vertex expansion / vertex pair contraction.
- A new topological data structure for non-regular, non-manifold meshes, which scales to the manifold case with a small overhead. This structure supports all vertex-based and triangle-based topological queries efficiently.
- Two algorithms for efficiently performing vertex-pair contraction and vertex insertion. These two algorithms are the basic ingredients for performing selective refinement based on a NMT. The extracted meshes are described by the topological structure proposed. Moreover, the algorithm which performs vertex-pair contraction can be used to produce a topological representation of a non-manifold mesh resulting from any iterative simplification algorithm based on the above operator [13, 17, 30].

The remainder of this paper is organized as follows. In Section 2, we discuss some related work. Section 3 introduces some notions used throughout the paper. In Section 4, we define the non-manifold MT in the general case. In Section 4.2, we discuss the selective refinement query, and we introduce the primitives needed for answering such query. In Section 5, we introduce a compact structure for a non-manifold MT built through vertex-pair contraction, and we evaluate its space requirements.

In Section 6, we define a new concise topological data structure for describing a triangle-segment mesh. In Section 7, we describe how mesh update primitives, that are involved in selective refinement, can be implemented on this data structure (more details are given in the Appendix). Section 8 contains some preliminary experimental results. Finally, in Section 9, we draw some concluding remarks and discuss future work.

2 Related Work

A considerable amount of work has been done in the last few years on simplification of triangle meshes, and a number of multi-resolution data structures have been proposed (see, e.g., Garland [16] for a survey). Here, we briefly review work related to our contribution, i.e., multi-resolution models for triangle meshes, and topological data structures for non-manifold objects.

Multi-resolution models for triangle meshes. In the literature there are several proposals for multi-resolution models based on manifold triangle meshes. There has been a lot of work on regular meshes based on nested recursive subdivisions for terrain modeling [11, 14, 25] and for subdivision surfaces [23, 35, 39]. Multi-resolution models for irregular meshes are structured as directed acyclic graphs [6, 7, 19, 31], or similar structures [37], or as forests of binary trees of vertices [22, 27], sometimes enriched with vertex labeling [13]. The multi-resolution structures proposed in [13, 27] for view-dependent rendering support non-manifold meshes (also non-regular meshes in [27]), but they extract just collections of cells, i.e., the mesh is described by a data structure without any adjacency information. Progressive Simplicial Complexes [30] are progressive models which do not support selective refinement. Moreover, an extracted mesh is not encoded in a topological data structure, but as a collection of cells.

Topological data structures for non-manifold meshes. The first proposal for a topological data structure for Boundary Representation (BRep) of non-manifold objects is the *radial-edge structure* by Weiler [36]. This structure has been specialized in [28] to the case of two-dimensional simplicial meshes. In [20], Gursoz et al. propose a vertex-based data structure, called the *tri-cyclic cusp* structure, which extends the radial-edge structure by keeping also inclusion relations between the local neighborhoods of a vertex. A similar structure has been introduced by Yagamuchi and Kimura [38]. Such structures have been designed to support an efficient navigation on the BRep without considering storage costs.

A more compact data structure, called the *partial entity structure* [24] has been recently proposed, which has been shown to require half of the space of the radial-edge structure. In [33], a powerful model for describing non-manifold, non-regular d-dimensional objects with possibly incomplete boundaries, called *Selective Geometric Complexes (SGC)*, has been proposed. The underlying data structure is an incidence graph [12], which requires less space than the above structures, but does not contain ordering information which are exploited by structure traversal algorithms.

A compact edge-based data structure for non-manifold, but regular triangle meshes is presented by Campagna et al. [1], called the directed-edges structure, which scales well to manifold meshes, since it has been developed under the assumption that the number of non-manifold entities is typically negligible compared to the complexity of the whole mesh.

An alternative approach to the design of non-manifold data structures consist of decomposing a non-manifold object into simpler and more manageable parts [10, 15, 18]. In [18], the idea of cutting a two-dimensional non-manifold complex into manifold pieces is used in order to develop a geometric compression algorithm. In [9], a decomposition of d-dimensional simplicial complexes into simpler components is defined, which can be exploited to define a scalable data structure for such complexes in three and higher dimensions.

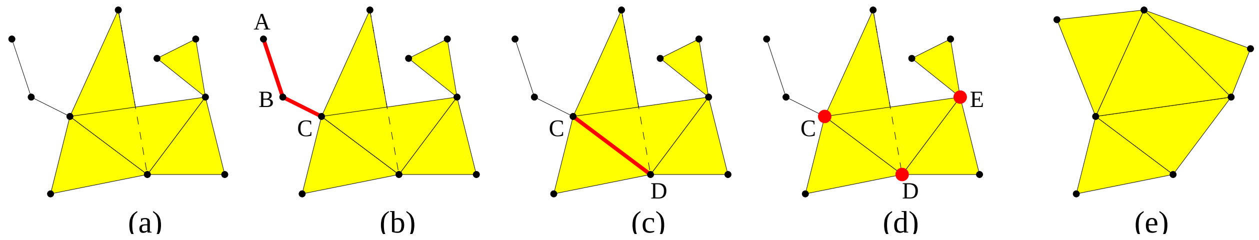


Figure 1: (a) A triangle-segment mesh; (b) (A, B) and (B, C) are wire-edges; (c) (C, D) is a non-manifold edge; (d) C, D, and E are non-manifold vertices. (e) A manifold triangle-segment mesh.

3 Background Notions

A *k-dimensional simplex* σ is the locus of points of $\mathbb{E}^3$ that can be expressed as the convex combination of $k+1$ affinely independent points, called the *vertices* of σ. Points, segments, and triangles are 0-, 1-, and 2-dimensional simplexes, respectively. A simplex σ' is called a *proper face* of another simplex σ if the vertices of σ' are a proper subset of those of σ.

A *two-dimensional simplicial complex* is a finite set Σ of k-dimensional simplexes, with $k \in \{0, 1, 2\}$, with the following properties: each pair of simplexes σ_1, $\sigma_2 \in \Sigma$ have disjoint interiors; and for each $\sigma \in \Sigma$, all proper faces of σ are in Σ.

A simplex $\sigma \in \Sigma$ is called a *top simplex* if it is not a proper face of any other simplex of Σ. We call a *triangle-segment mesh* a connected two-dimensional simplicial complex. With the exception of the degenerate case of a complex formed by just a single vertex, the top simplexes of a triangle-segment mesh are either triangles or segments, which motivates the name. Two examples of meshes, one non-regular non-manifold and the other manifold, are shown in Figure 1a and 1e.

An edge e of a mesh Σ is called a *wire-edge* if no triangle $t \in \Sigma$ exists such that e is an edge of t, i.e., if e is a top-edge (see Figure 1b). Otherwise, e is called a *triangle-edge*. A mesh is completely described by the set of its triangles plus the set of its wire-edges. We define the *size* of a mesh as the number of its triangles plus the number of its wire-edges.

A triangle-edge is a *manifold* edge if it has one or two incident triangles; if it has three or more incident triangles, then it is a *non-manifold* edge (see Figure 1c). A vertex v is a *manifold* vertex if one of the following conditions holds: at most two wire-edges are incident at v, and no triangle is incident at v; or no wire-edge is incident in v, and its incident triangles form a single fan, which may be either open or closed (this implies that all edges incident at v are manifold). Otherwise it is a *non-manifold* vertex (see Figure 1d). A *manifold triangle mesh* is a mesh in which there are no wire-edges, and all edges and vertices are manifold (see Figure 1e).

We consider the following parameters for a mesh Σ:

- n denotes the number of vertices,
- l denotes the number of wire-edges,

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

Table 1: Symbols that refer to meshes and NMTs.

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

Quantities l, c and a give a measure of the degree of "non-manifoldness" of mesh Σ. In a manifold triangle mesh, $l = c = a = 0$, and $m < 2n$. Usually, meshes that are encountered in the applications are "nearly" manifold, i.e., $l, c, a << m$, and $m \approx 2n$ (this latter equality being a consequence of the Euler formula). Table 1 summarizes the meaning of symbols defined above, as well as the meaning of other symbols defined in Section 4.

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

4 A Multi-resolution Model for Triangle-Segment Meshes

In this section, first we describe the Non-manifold Multi-Tessellation (NMT), which is a natural extension to the non-manifold domain of the model that we developed for the manifold case in [31] and our subsequent work. Next, we describe the basic structure of an algorithm that performs *selective refinement* on a NMT, which is also a natural extension of an algorithm we presented in [6].

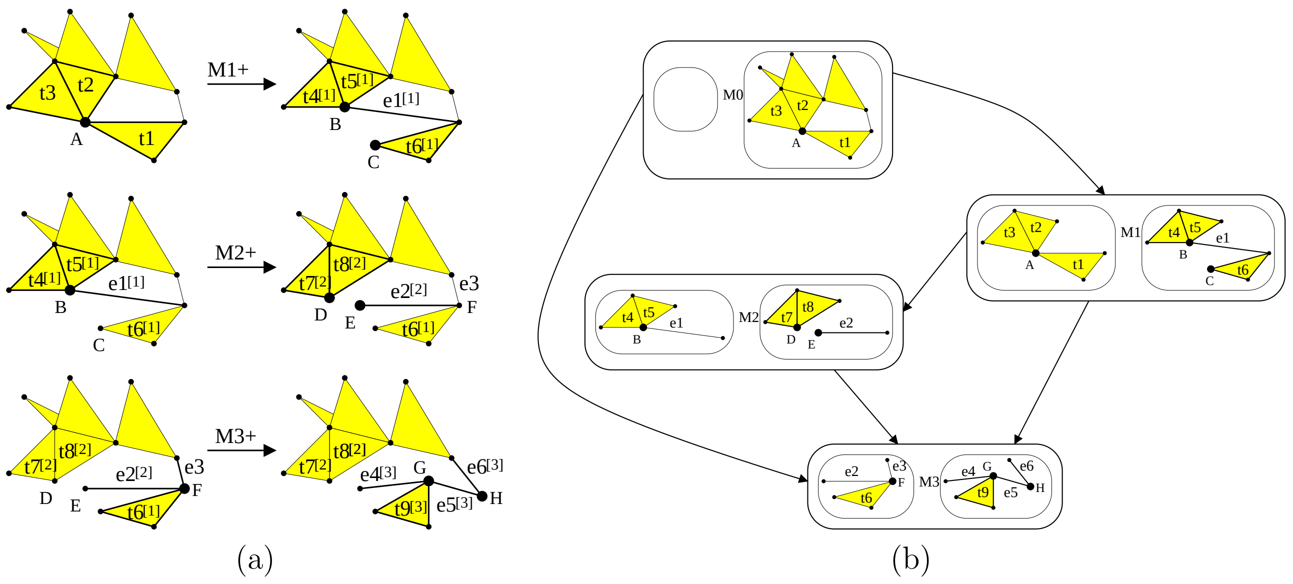


Figure 2: (a) A mesh refined by a sequence of three modifications. The elements (triangles and wire-edges) involved in each modification are drawn with thick edges. The elements created by a modification are labelled with its number between square brackets. (b) The corresponding NMT DAG. Each node contains the pair of sets of cells that defines the corresponding refinement and coarsening modification.

4.1 Non-Manifold Multi-Tessellation

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

We focus on modifications that change the size of a mesh by either increasing it (*refinement*), or decreasing it (*coarsening*). We use the notation M^+ for a refinement modification, and M^- for its inverse, i.e., a coarsening modification. Examples of refinement modifications are shown in Figure 2a.

Let $M_1^+, M_2^+, \ldots, M_h^+$ be a sequence of modifications that progressively refine a coarse mesh Σ_0 into a mesh Σ_h at full resolution (see Figure 2a). Note that the reversed sequence $M_h^-, \ldots, M_2^-, M_1^-$ progressively coarsens Σ_h into Σ_0. A multi-resolution mesh encompasses all meshes that can be constructed from such modifications. This is possible by defining a *partial order* that describes *mutual dependencies* between pairs of modifications.

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

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

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

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

The partial order can be described as a Directed Acyclic Graph (DAG), where arcs connect pair of nodes satisfying condition (1) above. By convention, if $M_i \prec M_j$, then the corresponding arc is directed from M_i to M_j (see Figure 2b). Mesh Σ_h obtained by applying all modifications, in any total order consistent with $\prec$, is called the *reference mesh,* and it is the mesh at the highest possible resolution that can be obtained from $\mathcal{M}$.

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

4.2 Selective Refinement on a NMT

A *selective refinement* query on a multi-resolution model consists of extracting a mesh, which satisfies some application-dependent requirements, such as approximating a spatial object with a certain accuracy which can be either uniform, or variable in space. In a selective refinement query, a multi-resolution mesh is used as a virtual mesh simplification algorithm.

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

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

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

Selective refinement can be performed by traversing the DAG describing a NMT and constructing a closed subset S of nodes, and its associated mesh Σ_S. An incremental algorithm was proposed in [6] for the manifold case, which finds a solution to a new query by starting

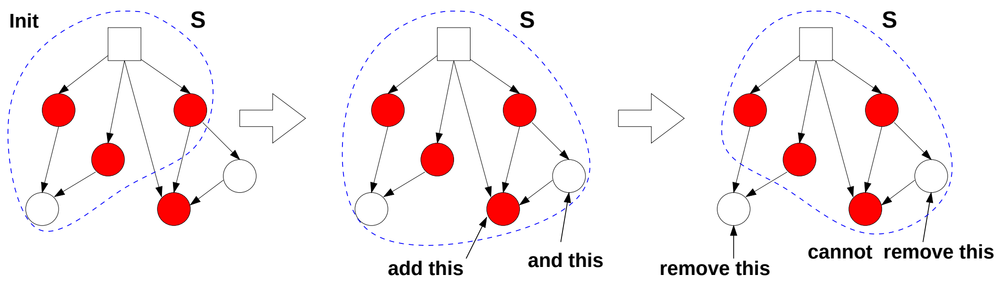


Figure 3: Selective refinement with an incremental approach. The square denotes the node M_0 corresponding to the base mesh. White nodes satisfy the function τ, dark nodes do not. The dashed line encloses the current set S.

from the solution to a previous query. This same algorithm can be applied also to a NMT, upon implementation of suitable data structures and related methods.

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

- A node M which is the destination of a front arc is added to S (M^+ is done on Σ_S) if $\tau(M) =$ *false* (i.e., if M is necessary to achieve the desired resolution). In order to maintain S a closed set, every modification M', such that $M' \notin S$, and $M' \prec M$, is recursively added to S before adding M.

- A node M which is the source of a front arc is deleted from S if it $\tau(M) =$ *true*, (i.e., if M^+ is not necessary). However, node M cannot be deleted if S contains some modification which depends on M. In this case, set $S \setminus M$ would not be closed.

A working example is shown in Figure 3.

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

1. *Insertion_Test*: decide if a node $M \notin S$ can be added to S while maintaining S a closed set, i.e., if all nodes preceding M in the dependency relation are in S;

2. *Removal_Test*: decide if a node $M \in S$ can be removed from S while maintaining S a closed set, i.e., if all nodes that depend on M are not in S;

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

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

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

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

5 A Compact Data Structure for a NMT

We consider now a NMT constructed by progressively coarsening the reference mesh Σ_h through a sequence of vertex-pair contractions [13, 30]. A *vertex-pair contraction* M^- identifies two vertices v'_M and v''_M of a mesh (possibly connected by an edge e_M) into a unique vertex v_M. All edges and triangles incident at v'_M and/or in v''_M become incident into v_M. Triangles incident at edge e_M (if any) degenerate to edges. The inverse modification M^+ of a vertex-pair contraction M^- is called *vertex expansion*.

The modifications M_i^- shown in Figure 2 are vertex-pair contractions: M_3^-, M_2^-, M_1^- contract vertices G, H to F; D, E to B; and B, C to A, respectively.

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

5.1 Encoding Nodes

The root node of the NMT, M_0, is encoded by a topological data structure representing the base mesh Σ_0, which will be described in the Section 6. Every other node represents a vertex-pair contraction, and its related vertex expansion.

Let M be a node such that modification M^- is a vertex-pair contraction, and modification M^+ is a vertex expansion. Following Popovic and Hoppe [30], we represent a node M by storing:

- Offset vectors to retrieve the coordinates of v'_M and v''_M from the coordinates of v_M, and vice versa. In general, we need two offset vectors $v'_M - v_M$ and $v''_M - v_M$. If v_M is chosen to be the midpoint of segment $v'_M v''_M$, one offset vector is sufficient.
- One bit EDGEFLAG to indicate whether an edge e_M connecting v'_M and v''_M exists.
- For each edge and triangle incident at v_M, two bits SPLITFLAG, to indicate how the edge or triangle is modified when expanding v_M (see Figure 4):
  - 0 if the edge or triangle becomes incident in v'_M, and not with v''_M (see Figure 4a);
  - 1 if the edge or triangle becomes incident in v''_M, and not with v'_M (see Figure 4b);
  - 2 if the edge or triangle is split into two edges or triangles, one incident in v'_M and the other one in v''_M (see Figure 4c);
  - 3 (only possible for edges, if EDGEFLAG= 1), if the edge is split into two edges and a new triangles is created which is incident in both v'_M and v''_M (see Figure 4d).

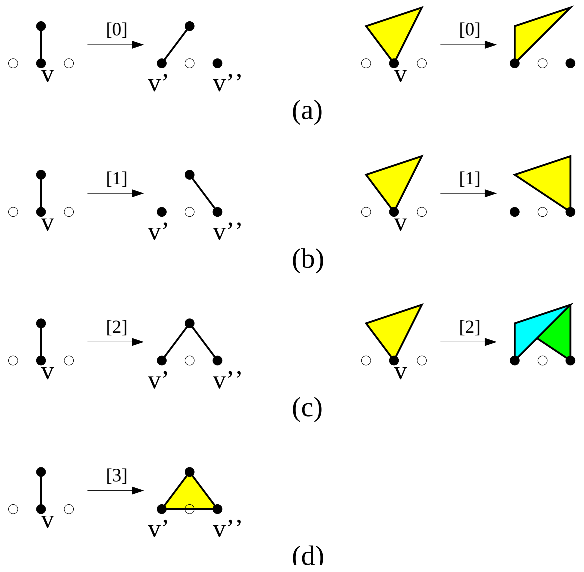


Figure 4: SPLITFLAG for an edge e and for a triangle t.

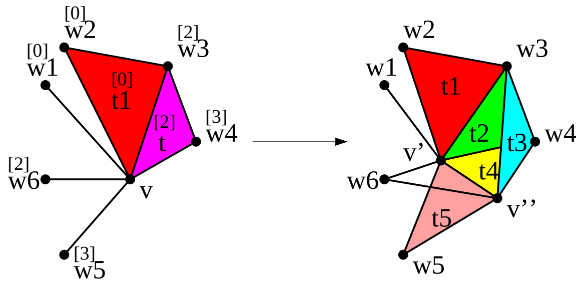


Figure 5: An example of using SPLITFLAGs in a vertex expansion. See text for explanation.

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

Figure 5 shows the use of the SPLITFLAGs in a vertex expansion, according to the convention. Vertices w_1, w_2 have SPLITFLAG$= 0$, thus edges (w_1, v) and (w_2, v) become (w_1, v') and (w_2, v'), respectively. Vertices w_3, w_6 have SPLITFLAG$= 2$, thus edges (w_3, v) and (w_6, v) become a pair of edges (w_3, v'), (w_3, v'') and (w_6, v'), (w_6, v''), respectively. Vertices w_4, w_5 have SPLITFLAG$= 3$, thus edges (w_4, v) and (w_5, v) become triangles $t_4 = (w_4, v', v'')$ and $t_5 = (w_5, v', v'')$, respectively. Triangle $t_1 = (w_2, w_3, v)$ has SPLITFLAG$= 0$, thus it becomes triangle (w_2, w_3, v'). Triangle $t = (w_2, w_3, v)$ has SPLITFLAG$= 2$, thus it becomes two triangles $t_2 = (w_2, w_3, v')$ and $t_3 = (w_2, w_3, v'')$.

The offset vectors for the coordinates are three real-valued numbers, requiring 12 bytes. The SPLITFLAGs are maintained in an array with the same sorting conventions as in [30] (the edges occupy the first positions, followed by the triangles). Each vertex in an NMT is labeled with a unique integer number (as explained in Section 5.2), and edges and triangles in the array are sorted lexicographically based on the labels of their vertices.

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

The total space complexity for storing all h modifications is thus equal to $12h$ bytes for offset vectors, and $(1 + 2D)h$ bits for the EDGEFLAGs and the SPLITFLAGs, where D is the average value of $\deg(v_M)$. The number h of modifications is bounded from above by the number n of vertices in the reference mesh Σ_h. The total cost for representing modifications is equal to $12n+8n = 20n$, by assuming for D a (pessimistic) upper bound of 31.5. The array of SPLITFLAGs could be further compressed by taking into account impossible configurations, as done in [30].

5.2 Encoding Dependencies

Since an explicit encoding of the DAG describing the partial order could be too expensive, we encode it implicitly through a mechanism proposed by El-Sana and Varshney [13]. This mechanism is based on a binary forest of vertices, and on a vertex enumeration scheme.

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

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

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

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

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

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

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

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

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

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

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

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

The correctness of Rule 3 follows immediately from Rule 2.

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

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

Following El-Sana and Varshney [13], the working data structure which represents the current mesh Σ_S is augmented by maintaining two more (small) integers for each vertex v, in order to speed-up the tests involved in primitives *Insertion_Test* and *removal_Test*. The two integers are the maximum vertex label of vertices adjacent to v in Σ_S, and the minimum vertex label of parents of vertices adjacent v in Σ_S. Such additional information need not be maintained in the output data structure, thus it is stored in a separate array. In this way, the implementation of *Insertion_Test* and *Removal_Test* does not require accessing all adjacent vertices on the current mesh. Rule 1 reduces to testing whether or not the stored minimum labels on parents of incident triangles is larger than the label of v. Rule 2 reduces to testing whether or not the stored minimum label on v is greater than the label of v.

6 A Data Structure for Triangle-Segment Meshes

In this section, we describe a new data structure for encoding a triangle-segment mesh as defined in Section 3. This data structure is used to encode the root mesh in the NMT, as well as the working structure for the current mesh, and the output structure produced by the selective refinement algorithm.

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

- *Vertex-Vertex* (VV) relation: it associates a vertex v with all its adjacent vertices. For clarity, we split such relation into two parts:

  - relation VV^{wir}, which contains those vertices adjacent to v through a wire-edge;
  - relation VV^{trg}, which contains those vertices adjacent to v through a triangle-edge.

  Both VV^{wir} and VV^{trg} are symmetric relations: $v' \in \mathrm{VV}^{wir}(v)$ if and only if $v \in \mathrm{VV}^{wir}(v')$, and the same holds for VV^{trg}.

- *Vertex-Triangle* (VT) relation: it associates a vertex v with all its incident triangles.

- *Triangle-Vertex* (TV) relation: it associates a triangle t with its three vertices v_1, v_2, v_3.

  Relation VT and TV are mutually inverse: $t \in \mathrm{VT}(v)$ if and only if $v \in TV(t)$.

- *Triangle-Triangle* (TT) relation: it associates a triangle t with three pairs of triangles, one for each edge e of t. Let v and w be the two endpoints of e. Then relation $\mathrm{TT}(t, v, w) = (t_1, t_2)$, where t_1 is the first triangle incident at edge e, that is encountered from t in counterclockwise order, and t_2 is the first one that is encountered from t in clockwise order, considering edge e as oriented from v to w (see Figure 6). If t has no adjacent triangles along edge e, then $t_1 = t_2 = t$.

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

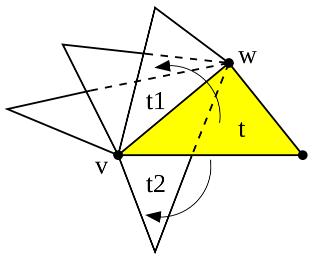


Figure 6: Relation $\mathrm{TT}(t, v, w)$ for a triangle t and edge $e = (v, w)$ of t: the triangles belonging to the relation are t_1 and t_2.

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

If we consider the set $\mathrm{VT}(v)$ of triangles incident in a vertex v, and the adjacency relations existing among the triangles of $\mathrm{VT}(v)$ along the edges incident in v, then set $\mathrm{VT}(v)$ is partitioned into one or more edge-connected components. For instance, for vertex E in Figure 1d, $\mathrm{VT}(E)$ consists of two edge-connected components.

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

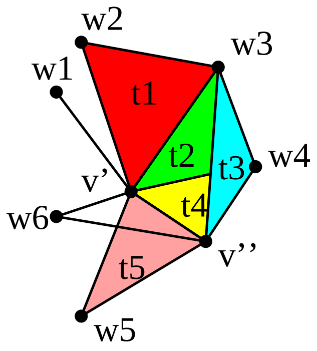
We have designed our data structure for triangle-segment meshes with a negligible overhead with respect to standard data structures for manifold triangle meshes, when it is used to represent a manifold triangle mesh. In a manifold triangle mesh, for each vertex v, relation $\mathrm{VV}^{wir}(v)$ is empty,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

For each vertex v, the data structure stores:

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

For each triangle t, the data structure stores:

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



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

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

Figure 7: A triangle-segment mesh, and the tabular representation of the data structure encoding it.

– if e is manifold, link to one of such triangles (since they are the same).

Figure 7 shows an example of the data structure. Note that, if $\mathrm{TT}(t, v_i, v_{i+1})$ is stored, then $\mathrm{TT}(t, v_{i+1}, v_i)$ is also virtually stored (it is sufficient to reverse the triangle pair).

6.1 Retrieval of Non-Stored Relations

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

In order to implement the basic operations *Mesh_Refinement* and *Mesh_Coarsening* for selective refinement, as explained in Section 7, we need retrieving relations VV^{trg} and VT for a given vertex v, and retrieving the cycle of triangles incident in an edge e, given a triangle incident in e.

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

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

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

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

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

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

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

The process stops when triangle t is encountered again.

6.2 Implementation and Space Requirements

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

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

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

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

- the three coordinates of v, requiring 12 bytes.
- a flag (on two bits) containing:
  - 0 if v is manifold and just triangles are incident in v (no wire-edges);
  - 1 if just wire-edges are incident in v (no triangles);
  - 2 if v is non-manifold and just triangles are incident in v (no wire-edges);

  - 3 if both triangles and wire-edges are incident in v.

- an index (on 4 bytes) with the following meaning, depending on the flag:
  - if 0: pointer to the unique triangle in $\mathrm{VT}^*(v)$;
  - if 1: pointer to a linked list containing $\mathrm{VV}^{wir}(v)$;
  - if 2: pointer to a linked list containing $\mathrm{VT}^*(v)$;
  - if 3: pointer to the first element of an array of two linked lists containing $\mathrm{VV}^{wir}(v)$ and $\mathrm{VT}^*(v)$, respectively; the same pointer shifted by one provides the address of the second element of the array.

  Each element of list $\mathrm{VV}^{wir}(v)$ is a pair containing the index of one vertex w and a pointer to the position of v in $\mathrm{VV}^{wir}(w)$. This latter pointer is not necessary to support mesh traversal, but it is used to improve the performance of algorithms that update the data structure, which will be described in Section 7.

For each triangle t, we store:

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

Linked lists have been used in order to allow performing vertex-pair contractions and vertex expansions on-line on the data structure. However, such lists are on average composed of very few elements (in our experiments, the average length is very close to one, and the maximum length never exceeds four). Thus, the computational overhead involved by managing linked lists is negligible.

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

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

Thus, the total space complexity of the data structure is equal to $16n+24m+24l+8c+4a$ bytes and $2n+3m$ bits. If the mesh is manifold, this reduces to $16n+24m \simeq 16n+48n = 64n$ bytes and $2n+6n = 8n$ bits $= n$ bytes, i.e., $65n$ bytes in total. Thus, the overhead introduced by the above data structure, when it is used to encode a manifold triangle mesh, is just one byte per vertex.

Our data structure is more compact than other structures proposed for non-manifold meshes: its storage cost is about one order of magnitude lower than the cost of the radial-edge data structure; one fourth of the cost of the directed-edge data structure; and about half the cost of the incidence graph. The main reason is that edges are not represented, and each other entity is represented only once. Our data structure supports traversal algorithm for evaluating vertex- and triangle-based relations that are more efficient than those supported by the incident graph, and as efficient as those supported by the other data structures. The main drawback of our data structure is that it cannot support attribute information associated with edges, since edges are just implicitly represented. For the same reason, traversal algorithms for evaluating edge-based relations are suboptimal, since they require traversing the stars of endpoints. The interested reader in referred to [4] for a more detailed comparison of non-manifold data structures.

7 Implementation of mesh update operations

In this section, we explain how to implement the basic operations for selective refinement defined in Section 4.2 on a current mesh Σ_S encoded with the data structure described in Section 6.

Implementations of operations *Insertion_Test*, *Removal_Test* and *Dependencies_Retrieval* follow immediately from the mechanism described in Section 5.2. To this aim, the data structure encoding NMT dependencies provides parent-child relations between vertices, and the topological data structure encoding the current mesh provides vertex-adjacency relations.

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

7.1 Performing vertex expansions

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

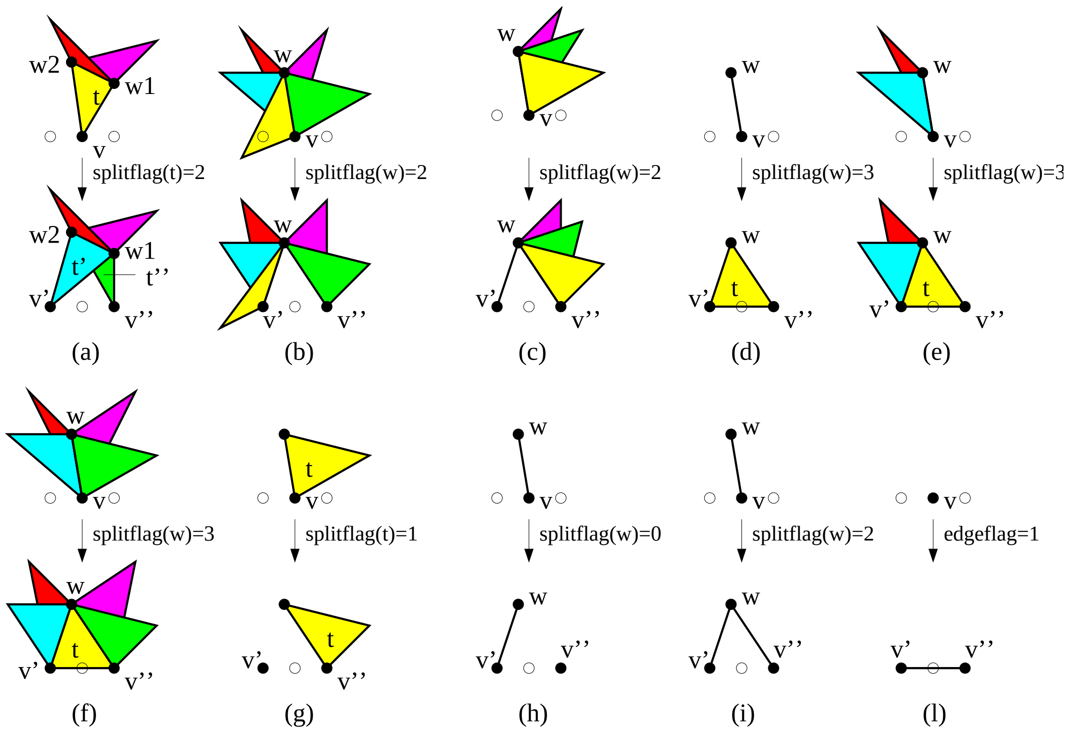


Figure 8: The possible configurations in vertex expansion.

specified in the codes associated with the corresponding modification M. The phases of the algorithm are described below.

- **Initialization.** Create the two new vertices v' and v'', retrieve relations VT(v) and VV$^{trg}(v)$, and build some auxiliary data structures that will be used later.

- **Duplicate coincident triangles.** Process each triangle $t' = (v, w_1, w_2) \in$ VT(v), such that SPLITFLAG$(t') = 2$, and transform it into two triangles incident in v' and in v'', respectively (see Figure 8a). Update relation TT at the edges of such triangles. After this phase, each pair of edges (v', w_i) and (v'', w_i) is still considered as one edge (w_i, v), having a unique cycle of triangles around it. Thus, the two new triangles temporarily coexist in the same cycle around (w_i, v), that will be later separated into two distinct cycles around edge (v', w_i) and (v'', w_i).

- **Separate coincident triangle-edges.** Process all triangle-edges (w, v) with SPLITFLAG$(w) = 2$ and transform each of them into two edges incident in v' and in v'', respectively (see Figure 8b,c). Moreover, update relation TT in order to separate the cycle of triangles around (w, v) into two cycles around (w, v') and (w, v''), respectively. If one of such cycles is empty, then update relation VV^{wir} in order to introduce a new wire-edge.

- **Expand wire-edges to triangles.** Process al vertices $w \in \mathrm{VV}^{wir}(v)$ with SPLITFLAG$(w) = 3$, and, for each of them, create a new triangle $t = (w, v', v'')$ (see Figure 8d).

- **Expand triangle-edges to triangles.** Process all vertices $w \in \mathrm{VV}^{trg}(v)$ with SPLITFLAG$(w) = 3$ and, for each of them, create a new triangle $t = (w, v', v'')$ (see Figure 8e,f). Moreover, update relation TT in order to separate the cycle of triangles around edge (w, v) into two cycles around (w, v') and (w, v''), respectively, and insert t in both cycles.

- **Update triangles.** In any triangle $t \in \mathrm{VT}(v)$, replace v with either v' or v'', depending on SPLITFLAG(t) (see Figure 8g).

- **Update wire-edges.** Examine the vertices in $\mathrm{VV}^{wir}(v)$ and update relation VV^{wir} in such a way to replace each wire-edge (w, v) with either (w, v'), or (w, v''), or both, depending on SPLITFLAG(w) (see Figure 8h,i). We also set wire-edge (v', v'') if EDGE-FLAG$= 1$ (see Figure 8l).

- **Adjust partial VT relation.** Check $\mathrm{VT}^*(v')$ and $\mathrm{VT}^*(v'')$, and eliminate any redundant triangle from them. This involves finding the edge-connected components of triangles incident in v' (and in v''), and then keep just one representative triangle for each component.

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

7.2 Performing vertex-pair contractions

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

- **Initialization.** Create vertex v and build some auxiliary structures that will be used later.

- **Contract triangles to edges.** Delete each triangle t belonging to $\mathrm{VT}(v') \cap \mathrm{VT}(v'')$ from the mesh (see Figure 8e,f). If necessary, replace t with a wire-edge, by updating relation VV^{wir} (see Figure 8d). Triangles to be deleted are recognized because they contain both vertices v' and v''.

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

Table 2: Sizes of datasets and of corresponding NMTs.

- **Glue triangle-edges.** Consider each pair of triangle-edges (w, v') and (w, v'') and merge the two cycles of triangles incident in them into one cycle around edge (w, v), by updating relation TT (see Figure 8b).
- **Glue triangle- and wire- edges.** Consider each pair of edges (w, v') and (w, v''), where one is a triangle- and the other one is a wire-edge, and delete the wire-edge by updating relation VV^{wir} (see Figure 8c).
- **Glue coincident triangles.** Consider each pair of triangles $t' = (w_1, w_2, v')$ and $t'' = (w_1, w_2, v'')$ and glue them into one triangle (w_1, w_2, v) (see Figure 8a). By convention, t' survives and is updated, while t'' is deleted. The triangles to be glued are found by checking the pairs of triangles that are mutually related through TT at an edge which is not incident v' or v''.
- **Update wire-edges.** Update relation VV^{wir} in order to transform wire edges incident in v' and v'' into wire-edges incident in v (see Figure 8h). This includes merging two wire-edges (w, v') and (w, v'') into one wire-edge (w, v) (see Figure 8i).
- **Update triangles.** Replace v' and v'' with v in the TV relation of all surviving triangles of $\mathrm{VT}(v') \cap \mathrm{VT}(v'')$ (see Figure 8g).
- **Adjust partial VT relation.** Eliminate redundant triangles from $\mathrm{VT}^*(v)$, by proceeding as explained in Section 7.1.

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

8 Results

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

We have considered a number of input meshes representing either natural or mechanical objects. We have selected objects that contain some elongated or thin parts, possibly meeting at small joints, because such configurations are suitable to be represented through non-manifold, non-regular meshes at low level of detail. Here, we present two examples of datasets: one representing a horse, and another representing a stool. The input datasets are depicted in Figures 9a and reffig:stoola. Their size and the size of the NMT built on each of them are reported in Table 2.

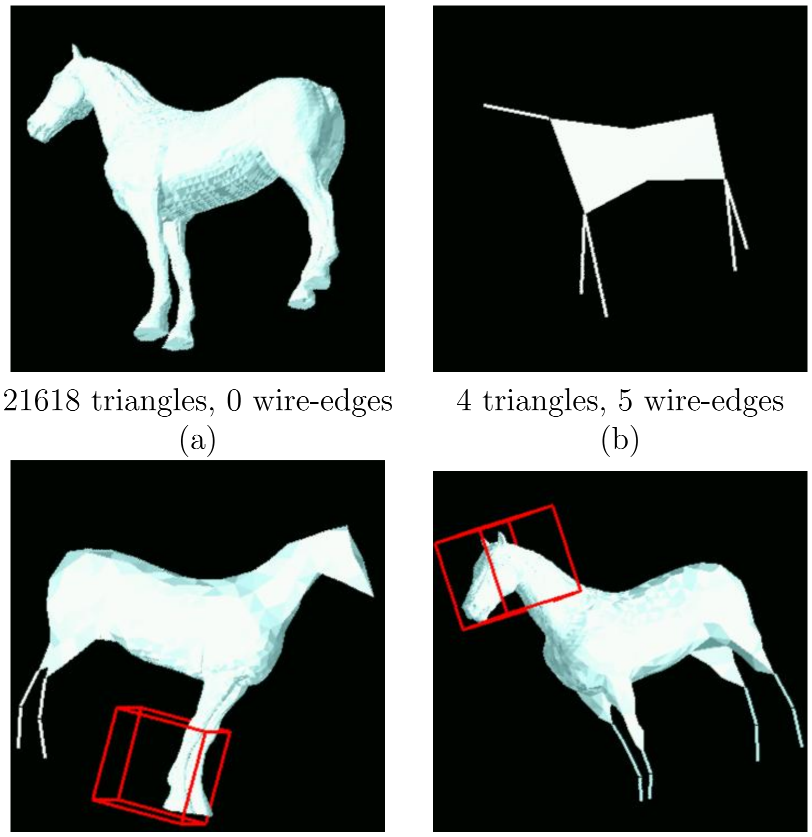
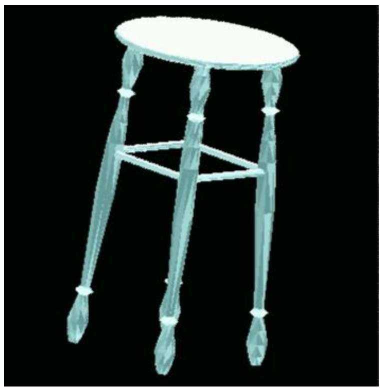

21618 triangles, 0 wire-edges
(a)
4 triangles, 5 wire-edges
(b)
3089 triangles, 4 wire-edges
(c)
9494 triangles, 10 wire-edges
(d)

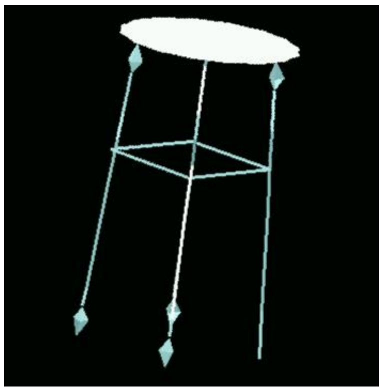
Figure 9: Some meshes meshes representing the horse at (a) the highest resolution, (b) the lowest resolution, (c) and (d) variable LOD.

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

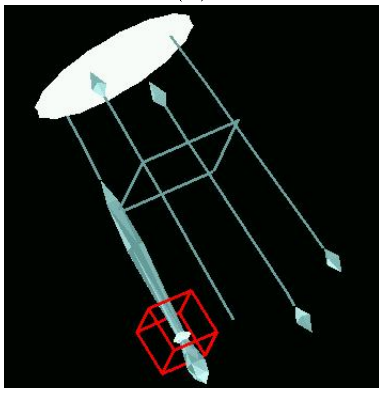
We run experiments that perform spatial selection queries, based on a box that moves through space containing the object. The Boolean function defining selective refinement requires that level of detail is maximum inside the box and arbitrarily low elsewhere. The box moves through space along a trajectory with a fixed step. At each step, selective refinement is applied starting at the mesh extracted at the previous position. We consider two trajectories: one is rectilinear and the other is random. Figures 9c, 9d, 10c, and reffig:stoold show screenshots of the experiments for the two datasets. For each dataset, trajectory and step



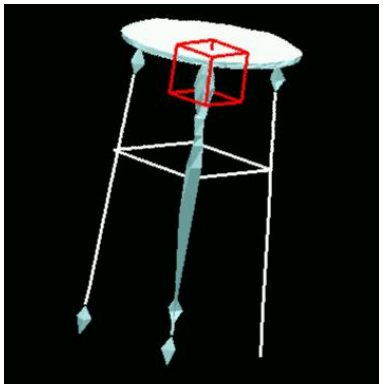
1788 triangles, 0 wire-edges
(a)



319 triangles, 25 wire-edges
(b)



208 triangles, 27 wire-edges
(c)



319 triangles, 25 wire-edges
(d)

Figure 10: Some meshes meshes representing the stool at (a) the highest resolution, (b) the lowest resolution, (c) and (d) variable LOD.

we counted the following quantities:

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

In Table 3 we report the average and standard deviation of such quantities computed on all frames for the two datasets and the two trajectories.

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

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

(a)

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

(b)

Table 3: Statistics on experiments (horse: 40 frames; stool: 10 frames). (a) Shows the number of NMT nodes in the closed set S corresponding to the current mesh Σ_S, and the number of triangles and wire-edges in Σ_S. (b) Shows the number of nodes removed from and inserted in S. For each item, the average and the standard deviation over all frames are reported.

Since the performance of algorithms described in subsections 7.1 and 7.2 depends on the manipulation of linked lists, we have also counted, for each collapse operation (hence for each node in the MT), the number of wire-edges and the number of fans of triangles incident at the vertex resulting from a collapse. Our experiments have shown that both such numbers (hence the length of linked lists) never exceed four. Moreover, the average number of fans of triangles incident at a vertex is less than 1.2; only 2% of vertices have incident wire edges, and the average number of edges incident at one such vertex is less than 1.25. Note that these statistics also indicate a small degree of "non-manifoldness" of meshes encoded in (and extracted from) a NMT, thus motivating our approach of working with structures that scale well to the manifold case.

9 Concluding Remarks

We have addressed the problem of representing and processing non-regular, non-manifold simplicial meshes, that we call triangle-segment meshes, at different levels of detail. Such meshes are used to describe spatial objects consisting of parts of mixed dimensions, and with a complex topology. We have described a model for multi-resolution representation of triangle-segment meshes, the NMT, and we have described a compact data structure for an instance of this model built through vertex-pair contractions.

We have proposed a new compact data structure for triangle-segment meshes, which represents both connectivity and adjacency information with a small memory overhead. We have shown that this data structure scales very well to the manifold case, since it costs only 1/64 more than the corresponding manifold structure encoding just triangle-vertex, triangle-triangle and partial vertex-triangle relations. From the proposed structure we can extract all vertex-based and triangle-based relations in optimal time, in particular the ones required by the algorithms for vertex-pair contraction and vertex expansion.

On the basis of the above structure and of efficient structure update algorithms, presented in Section 7, our approach is capable of generating a complete topological description of a mesh extracted through selective refinement, which is well-suited for analysis operations based on mesh navigation.

In the immediate future, we plan design more sophisticated simplification criteria to drive the construction of the NMT. By taking into account non-regular and non-manifold features, we believe we could highly improve the results of selective refinement. In fact, as we experimented in the manifold case [7], the shape the MT, and thus its power of selectivity, is greatly affected by the simplification strategy used. We also plan to study more sophisticated strategies to drive selective refinement (level-of-detail measures).

Further developments of the work presented in the paper are concerned with extending this approach to three and higher dimensions. The NMT as well as the data structure for describing its instance built through vertex-pair contraction can be extended without major difficulties. The crucial part is in defining a compact and scalable data structure for the current and the extracted mesh, which efficiently supports navigation and update operations. To this aim, we are currently working on the design of a scalable topological data structure for describing a non-manifold non-regular simplicial complex in arbitrary dimension [9].

Acknowledgements

This work has been partially performed while L. De Floriani has been visiting the Computer Science Department of the University of Maryland, College Park (USA). This work has been partially supported by a project funded by the Italian Ministry of University and Scientific and Technological Research on *Representation and Processing of Spatial Data in Geographic Information Systems,* and by a project funded by the Italian National Research Council on *Efficient modeling and transmission of three-dimensional scenes and objects* under program "Agenzia 2000", Contract N.CNRC00FE45_004.

References

[1] S. Campagna, L. Kobbelt, and H.-P. Seidel. Directed edges - a scalable representation for triangle meshes. *Journal of Graphics Tools*, 4(3), 1999.

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

[3] P. Cignoni, C. Montani, and R. Scopigno. A comparison of mesh simplification algorithms. *Computers and Graphics*, 22(1):37–54, 1998.

[4] L. De Floriani and A. Hui. Analysis and comparison of data structures for two-dimensional complexes embedded in 3d space. Technical report, Dept. ofComputer Science, University of Maryland, College Park (MD), 2002.

[5] L. De Floriani and P. Magillo. Multiresolution mesh representation: Models and data structures. In M.Floater, A.Iske, and E.Qwak, editors, *Tutorials on Multiresolution in Geometric Modelling*, pages 363–418. Springer-Verlag, 2002.

[6] L. De Floriani, P. Magillo, F. Morando, and E. Puppo. Dynamic view-dependent multiresolution on a client-server architecture. *CAD Journal*, 32(13):805–823, 2000.

[7] 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.

[8] 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*, pages 302–323. Springer-Verlag, 1997.

[9] L. De Floriani, M. M. Mesmoudi, F. Morando, and E. Puppo. Non-manifold decomposition in arbitrary dimension. In A. Braquelaire, J.-O. Lachaud, and A. Vialard, editors, *Discrete Geometry for Computer Imagery, Lecture Notes in Computer Science*, volume 2301, pages 69–80. Springer-Verlag, 2002.

[10] H. Desaulnier and N. Stewart. An extension of manifold boundary representation to r-sets. *ACM Transactions on Graphics*, 11(1):40–60, 1992.

[11] M. Duchaineau, M. Wolinsky, D.E. Sigeti, M.C. Miller, C. Aldrich, and M.B. Mineed-Weinstein. ROAMing terrain: Real-time optimally adapting meshes. In R. Yagel and H. Hagen, editors, *Proceedings IEEE Visualization'97*, pages 81–88, 1997.

[12] H. Edelsbrunner. *Algorithms in Combinatorial Geometry.* Springer-Verlag, Berlin, 1987.

[13] J. El-Sana and A. Varshney. Generalized view-dependent simplification. *Computer Graphics Forum*, 18(3):C83–C94, 1999.

[14] W. Evans, D. Kirkpatrick, and G. Townsend. Right-triangulated irregular networks. *Algorithmica*, 30(2):264–286, 2001.

[15] B. Falcidieno and O. Ratto. Two-manifold cell-decomposition of r-sets. *Comuter Graphics Forum (EUROGRAPHICS '92 Proceedings)*, 11(3):391–404, September 1992.

[16] M. Garland. Multiresolution modeling: Survey & future opportunities. In *Eurographics '99 – State of the Art Reports*, pages 111–131, 1999.

[17] M. Garland and P.S. Heckbert. Surface simplification using quadric error metrics. In *Computer Graphics Proceedings, Annual Conference Series (SIGGRAPH '97), ACM Press*, pages 209–216, 1997.

[18] A. Guéziec, F. Bossen, G. Taubin, and C. Silva. Efficient compression of non-manifold polygonal meshes. *Computational Geometry: Theory and Applications*, 14:137–166, 1999.

[19] A. Guéziec, G. Taubin, F. Lazarus, and W. Horn. Simplicial maps for progressive transmission of polygonal surfaces. In *Proceedings ACM VRML98*, pages 25–31, 1998.

[20] E.L. Gursoz, Y. Choi, and F.B. Prinz. Vertex-based representation of non-manifold boundaries. In *Geometric Modeling for Product Engineering*, pages 107–130, North Holland, 1990. Elsevier Science Publishers.

[21] T. He, L. Hong, A. Varshney, and S. Wang. Controlled topology simplification. *IEEE Transactions on Visualization and Computer Graphics*, 2(2):171–183, 1996.

[22] H. Hoppe. View-dependent refinement of progressive meshes. In *ACM Computer Graphics Proceedings, Annual Conference Series, (SIGGRAPH '97)*, pages 189–198, 1997.

[23] L. Kobbelt. $\sqrt{3}$ subdivision. In *Computer Graphics Proceedings, Annual Conference Series (SIGGRAPH 2000), ACM Press*, 2000.

[24] S. Lee and K. Lee. Partial entity structure: a fast and compact non-manifold boundary representation based on partial topological entities. In *Proceedings Sixth ACM Symposium on Solid Modeling and Applications*, Ann Arbor, Michigan, June 4-8 2001.

[25] 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 *ACM Computer Graphics (SIGGRAPH '96 Proceedings)*, pages 109–118, New Orleans, LA, USA, Aug. 6-8 1996. ACM Press.

[26] D. Luebke. A developer's survey of polygonal simlification algorithms. *IEEE Computer Graphics & Applications*, 21(3), May-June 2001.

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

[28] S. McMains and C. S. J. Hellerstein. Out-of-core build of a topological data structure from polygon soup. In *Proccedings Sixth ACM Symposium on Solid Modeling and Applications*, pages 171–182, Ann Arbor, Michigan (USA), June 6-8 2001.

[29] P.Lindstrom. Out-of-core simplification of large polygonal models. In *ACM Computer Graphics (SIGGRAPH 2000 Proceedings)*, pages 259–270, New Orleans, LA, USA, July 2000. ACM Press.

[30] J. Popovic and H. Hoppe. Progressive simplicial complexes. In *ACM Computer Graphics Proceedings, Annual Conference Series, (SIGGRAPH '97)*, pages 217–224, 1997.

[31] 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 appeared with title Variable Resolution Triangulations, *Computational Geometry*, 1998, 11(3-4):219-238.

[32] J. Rossignac and P. Borrel. Multi-resolution 3D approximation for rendering complex scenes. In B. Falcidieno and T.L. Kunii, editors, *Geometric Modeling in Computer Graphics*, pages 455–465. Springer Verlag, 1993.

[33] J.R. Rossignac and M.A. O'Connor. SGC: A dimension-independent model for pointsets with internal structures and incomplete boundaries. In J.U. Turner, M.J. Wozny, and K. Preiss, editors, *Geometric Modeling for Product Engineering*, pages 145–180, North Holland, Amsterdam, 1990. Elsevier Science Publishers.

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

[35] L. Velho, L. Henriquez de Figueredo, and J. Gomes. A unified approach for hierarchical adaptive tesselation of surfaces. *ACM Transactions on Graphics*, 4(18):329–360, 1999.

[36] K. Weiler. The radial edge data structure: A topological representation for non-manifold geometric boundary modeling. In J. E. M.J. Wozny and H.W. McLaughlin, editors, *Geometric Modeling for CAD Applications*, pages 3–36. North-Holland, 1988.

[37] 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.

[38] Y. Yamaguchi and F. Kimura. Nonmanifold topology based on coupling entities. *IEEE Computer Graphics and Applications*, 15(1), January 1995.

[39] D. Zorin, P. Schröder, and W. Sweldens. Interactive multiresolution mesh editing. In *Computer Graphics Proceedings, Annual Conference Series (SIGGRAPH '97), ACM Press*, pages 259–268, 1997.

A Appendix

In this appendix, we give more details on the refinement and coarsening operations for selective refinement, that we outlined in Section 7.

A.1 Vertex expansion

Vertex v is expanded to vertices v' and v''. For evaluating the time complexity we use two parameters: the cardinality N_{VT} of $\mathrm{VT}(v)$ and the cardinality N_{VV} of $\mathrm{VV}^{wir}(v)$. Note that the cardinality of $\mathrm{VV}^{trg}(v)$ has the same complexity order as N_{VT} because each triangle of $\mathrm{VT}(v)$ contributes with at most two vertices to $\mathrm{VV}^{trg}(v)$.

Initialization

- Retrieve all the triangles in $\mathrm{VT}(v)$ and store them in a list l_{VT}.
- Retrieve all the vertices in $\mathrm{VV}^{trg}(v)$ and store them in a list $l_{\mathrm{VV}trg}$, where each vertex w has a pointer to one triangle $t_w \in \mathrm{VT}(v)$, containing w among its vertices.
- Create a new vertex v' and a new vertex v''.
- Initialize relations VV^{wir} and VT^* as empty for v' and v''.
- For each triangle $t \in \mathrm{VT}^*(v)$:
  - if SPLITFLAG$(t) \neq 1$, then add t to $\mathrm{VT}^*(v')$;
  - otherwise find one triangle t' with SPLITFLAG$(t') \neq 1$, among the triangles belonging to the same edge-connected component of $\mathrm{VT}(v)$; if such a t' is found then add it to $\mathrm{VT}^*(v')$;
  - if SPLITFLAG$(t) \neq 0$, then add t to $\mathrm{VT}^*(v'')$;
  - otherwise find one triangle t' with SPLITFLAG$(t') \neq 0$, among the triangles belonging to the same edge-connected component of $\mathrm{VT}(v)$; if such a t' is found then add it to $\mathrm{VT}^*(v'')$;
- Initialize an empty structure l_{new} that will contain the new triangles which are created incident in edge (v', v''). Such structure can be a binary search tree built according to the counterclockwise radial order around (v', v'').

The time complexity of this phase is in $O(N_{\mathrm{VT}})$. While constructing $\mathrm{VV}^{trg}(v)$ with the algorithm described in Section 6.1, it is easy to associate one triangle t_w with each vertex w, without any additional cost. Finally, each edge-connected component of $\mathrm{VT}(v)$ is traversed at most twice, and such components form a partition of $\mathrm{VT}(v)$.

Duplicate coincident triangles

For each $t' \in l_{\mathrm{VT}}$ with SPLITFLAG$(t') = 2$:

- Create a new triangle t'' with $\mathrm{TT}(t'') = (v'', w_1, w_2)$. Replace v with v' in $\mathrm{TT}(t')$.
- Set SPLITFLAG$(t') = 0$ and SPLITFLAG$(t'') = 1$.
- Update relation TT at the edge (w_1, w_2) shared by t' and t'', in order to insert t'' in the cycle of triangles around (w_1, w_2). The correct position for t'' is either before or after t', and is found by looking at geometry.
- Update relation TT at the other two edges (v, w_i), $i \in [1, 2]$ (edges (v', w_i) and (v'', w_i) are still identified here), in order to insert t'' in the cycle around (v, w_i). t'' can be inserted either before or after t', the choice is not relevant because the cycle around (v, w_i) will be separated into two cycles later.
- Add t'' to l_{VT} in order to let it be processed in the next phases.

After this step, all SPLITFLAGs of triangles are in $\{0, 1\}$. The time complexity of this phase is in $O(N_{VT})$. This is due to the scan of list l_{VT}, while the operations at each triangle t' have a constant cost.

Separate coincident triangle-edges

For each vertex w in list $l_{\mathrm{VV}trg}$ with SPLITFLAG$(w) = 2$:

- Let t_w be the triangle associated with w in $l_{\mathrm{VV}trg}$.
- Starting from t_w, retrieve the cycle of triangles around edge (w, v).
- Update the relation $\mathrm{TT}(t, w, v)$ for the triangles t of such cycle in such a way to build two cycles, containing triangles with SPLITFLAG$= 0$ and 1, respectively.
- If the resulting cycle of triangles with SPLITFLAG$= 0$ is empty, then add w to $VV^{wir}(v')$ and v' to $VV^{wir}(w)$.
- If the resulting cycle of triangles with SPLITFLAG$= 1$ is empty, then add w to $VV^{wir}(v'')$ and v'' to $VV^{wir}(w)$.

The time complexity of this phase is in $O(N_{\mathrm{VT}})$. This includes the scan of all cycles of triangles around triangle-edges incident in v (because these cycles are formed by triangles of $\mathrm{VT}(v)$, and each triangle can take part in at most two cycles).

Expand wire-edges to triangles

For each vertex $w \in \mathrm{VV}^{wir}(v)$ with SPLITFLAG$(w) = 3$:

- Create a new triangle $t = (w, v', v'')$ and initialize its TT relations as pairs (t, t) (i.e., t has no adjacent triangles).
- If list l_{new} is not empty (i.e., t is not the first triangle being created), then update relation TT at edge (v', v'') for t and for the previously created triangles, in order to insert t in the cycle of triangles around (v', v''). The position of t in such cycle depends on geometry.
- Otherwise (i.e., t is the first triangle being created), add t to $\mathrm{VT}^*(v')$ and to $\mathrm{VT}^*(v'')$, because t is the only element of a new edge-connected component.
- Add t to l_{new}.
- Add t to $\mathrm{VT}^*(w)$.

The time complexity of this phase is in $O(N_{\mathrm{VV}})$ plus the cost of inserting the new triangles into the cycle of triangles around (v', v''), implemented by l_{new}. All other operations performed at a vertex w have a constant cost. The cost of operations of insertions into l_{new} is $O(A \log A)$ where $A < N_{\mathrm{VV}}$ is the number of triangles of the cycle. The logarithmic term is due to the search for the correct position of each new triangle, which can be done with a standard search structure built on-the-fly and discarded afterwards.

Expand triangle-edges to triangles

For each vertex $w \in l_{\mathrm{VV}trg}$ with SPLITFLAG$(w) = 3$:

- Create a new triangle $t = (w, v', v'')$ and initialize its TT relations as pairs (t, t) (i.e., t has no adjacent triangles).
- Let t_w be the triangle associated with w in $l_{\mathrm{VV}trg}$.
- Starting from t_w, retrieve the cycle of triangles around edge (w, v).
- Update the relation $\mathrm{TT}(t', w, v)$ for the triangles t' of such cycle in such a way to build two cycles, containing triangles with SPLITFLAG$= 0$ and 1, respectively.
- Add t to both resulting cycles. The position of t in each of them is determined based on geometry.
- If list l_{new} is not empty (i.e., t is not the first triangle being created), then update relation TT at edge (v', v'') in order to insert t in the cycle of triangles around (v', v''). The position of t in the cycle depends on geometry.
- add t to l_{new}.

The time complexity of this phase is in $O(N_{\mathrm{VT}} + A \log A)$, where A has the same meaning as in the previous phase. The motivations for such conclusions have already been explained while discussing previous phases.

Update triangles

For each triangle $t \in l_{\mathrm{VT}}$:

- if $\textsc{splitflag}(t) = 0$, then replace v with v' in $\mathrm{TT}(t)$;
- otherwise (i.e., if $\textsc{splitflag}(t) = 0$), replace v with v'' in $\mathrm{TT}(t)$;

The time complexity of this phase is in $O(N_{\mathrm{VT}})$.

Update wire-edges

- For each vertex $w \in \mathrm{VV}^{wir}(v)$:
  - If $\textsc{splitflag}(w) = 0$ or 2 then add w to $\mathrm{VV}^{wir}(v')$ and v' to $\mathrm{VV}^{wir}(w)$;
  - If $\textsc{splitflag}(w) = 1$ or 2 then add w to $\mathrm{VV}^{wir}(v'')$ and v'' to $\mathrm{VV}^{wir}(w)$;
  - Delete v from $\mathrm{VV}^{wir}(w)$;
- If $\textsc{edgeflag}= 1$ (i.e., (v', v'') is a wire-edge), then add v'' to $\mathrm{VV}^{wir}(v')$ and v' to $\mathrm{VV}^{wir}(v'')$.

The time complexity of this phase is in $O(N_{\mathrm{VV}})$. Note that deletion from VV^{wir} is done in constant time since $\mathrm{VV}^{wir}(v)$ contains the position of v into $\mathrm{VV}^{wir}(w)$.

Adjust partial VT relation

We scan $\mathrm{VT}^*(v')$ and, for each triangle $t \in \mathrm{VT}^*(v')$:

- if t is marked then remove t from $\mathrm{VT}^*(v')$;
- otherwise retrieve the edge-connected component of $\mathrm{VT}(v')$ containing t and mark all its triangles (including t).

A similar process is applied to $\mathrm{VT}^*(v'')$.

The time complexity of this phase is linear in the number of elements of $\mathrm{VT}(v')$ and $\mathrm{VT}(v'')$ because each edge-connected component of VT is retrieved only once, in linear time, and these components form a partition of VT. The sum of cardinalities of $\mathrm{VT}(v')$ and $\mathrm{VT}(v'')$ has the same order as $N_{\mathrm{VT}} + N_{\mathrm{VV}}$.

In summary, the time complexity of expanding vertex v is in $O(N_{\mathrm{VT}} + N_{\mathrm{VV}} + A \log A)$, where A is the number of triangles that are created incident at (v', v''). However, A can be considered a constant in practical cases.

A.2 Vertex-pair contraction

Vertices v' and v'' are contracted to vertex v. Parameter N_{VT} denotes the sum of the cardinalities of $\mathrm{VT}(v')$ and $\mathrm{VT}(v'')$, and parameter N_{VV} denotes the sum of the cardinalities of $\mathrm{VV}^{wir}(v')$ and $\mathrm{VV}^{wir}(v'')$.

Initialization

- Create vertex v.
- Initialize $\mathrm{VV}^{wir}(v)$ as empty.
- Initialize $\mathrm{VT}^*(v)$ as the union of $\mathrm{VT}^*(v')$ and $\mathrm{VT}^*(v'')$.
- Retrieve all the triangles in $\mathrm{VT}(v')$ and store them in a list l'_{VT}; retrieve all the triangles in $\mathrm{VT}(v'')$ and store them in a list l''_{VT}.
- Retrieve all the vertices in $\mathrm{VV}^{trg}(v')$ and store them in a list $l'_{\mathrm{VV}trg}$; retrieve all the vertices in $\mathrm{VV}^{trg}(v'')$ and store them in a list $l''_{\mathrm{VV}trg}$; each vertex w in each list has a pointer to one triangle t_w of $\mathrm{VT}(v')$ and $\mathrm{VT}(v'')$, respectively, containing w; triangle t_w is chosen in such a way that it does not contain both v' and v''. If a triangle satisfying this requirement does not exist, then a null triangle is stored.

The time complexity of this phase is in $O(N_{\mathrm{VT}})$. The motivations are similar to those explained for the initialization of the algorithm for vertex expansion.

Contract triangles to edges

We first scan list l'_{VT} and then l''_{VT}. For each triangle $t \in l'_{\mathrm{VT}}$ such that $\mathrm{VT}(t)$ contains both v' and v'':

- Let w be the third vertex of t;
- If t is not the only triangle around edge (w, v') then update relation TT at (w, v') in order to delete t from the cycle (it is sufficient to update the two triangles found in $\mathrm{TT}(t, w, v')$).
- If t is not the only triangle around edge (w, v'') then update relation TT at (w, v'') in order to delete t from the cycle (as above). Otherwise, t degenerates to a wire-edge, which will be handled in the next phase.
- delete t from l'_{VT}.

For each triangle $t \in l''_{\mathrm{VT}}$ such that $\mathrm{VT}(t)$ contains both v' and v'':

- delete t from l''.
- delete t from Σ.

The computational complexity is in $O(N_{\mathrm{VT}})$.

Glue triangle-edges

For each vertex $w \in l'_{\mathrm{VV}trg} \cap l''_{\mathrm{VV}trg}$:

- Let t_{w1} and t_{w2} be the triangle associated with w in $l'_{\mathrm{VV}trg}$ and in $l''_{\mathrm{VV}trg}$, respectively.
- Starting from t_{w1} and t_{w2}, find the cycles of triangles around edges (w, v') and (w, v''). If one of t_{w1} and/or t_{w2} is null, then consider an empty cycle around (w, v') and/or (w, v'').
- Update the TT relation in order to merge the two cycles of triangles, into one cycle around (w, v): whenever a triangle t_1 coming the first cycle is followed by a triangle t_2 coming from the opposite cycle, relations TT must be updated for t_1, t_2 and for their related triangles, at edge (w, v).
- If the resulting merged cycle is empty, then (w, v) is a new wire-edge, therefore add w to $\mathrm{VV}^{wir}(v)$ and w to $\mathrm{VV}^{wir}(w)$.
- Note that this last operation creates the wire edges for triangles that collapsed to wire-edges in the previous phase.

The intersection $l'_{\mathrm{VV}trg} \cap l''_{\mathrm{VV}trg}$, can be constructed from $l'_{\mathrm{VV}trg}$ and $l''_{\mathrm{VV}trg}$ in linear time by first marking vertices that belong to $l'_{\mathrm{VV}trg}$ and then collecting marked vertices from $l''_{\mathrm{VV}trg}$. The cycles of triangles around (w, v') and (w, v'') are constructed and merged in linear time in their size, which is bounded from above by the sum of cardinalities of $\mathrm{VT}(v')$ and $\mathrm{VT}(v'')$. Thus, the time complexity of this phase is in $O(N_{VT})$.

Glue triangle- and wire- edges

- For each vertex $w \in l'_{\mathrm{VV}trg} \cap \mathrm{VV}^{wir}(v'')$, delete w from $\mathrm{VV}^{wir}(v'')$.
- For each vertex $w \in l''_{\mathrm{VV}trg} \cap \mathrm{VV}^{wir}(v')$, delete w from $\mathrm{VV}^{wir}(v')$.

The time complexity of this phase is in $O(N_{\mathrm{VT}} + N_{\mathrm{VV}})$.

Glue coincident triangles

For each triangle $t'' \in l''_{\mathrm{VT}}$:

- Let w_1, w_2 be the vertices of t'' different from v'';
- Let $(t_1, t_2) = \mathrm{TT}(t'', w_1, w_2)$;
- if t_1 has v' or t_2 has v', then
  - update TT relations in order to delete t'' from the cycle of triangles around edge $(w_1 w_2)$, from the cycle around edge (w_1, v), and from that around (w_2, v);

  - delete t'' from $t'' \in l''_{\mathrm{VT}}$ in order to avoid processing it in next phases.
  - delete t.

The time complexity of this phase is in $O(N_{\mathrm{VT}})$.

Update wire-edges

- For each $v \in \mathrm{VV}^{wir}(v')$, add w to $\mathrm{VV}^{wir}(v)$, replace v' with v in $\mathrm{VV}^{wir}(w)$, and mark w.
- For each $v \in \mathrm{VV}^{wir}(v'')$, if w is not marked then add w to $\mathrm{VV}^{wir}(v)$, and replace v'' with v in $\mathrm{VV}^{wir}(w)$; otherwise simply delete v'' from $\mathrm{VV}^{wir}(w)$, and unmark w.

The time complexity of this phase is in $O(N_{\mathrm{VV}})$.

Update triangles

- For every triangle $t \in l'_{\mathrm{VT}}$, update $\mathrm{TT}(t)$ by replacing v' with v.
- For every triangle $t \in l''_{\mathrm{VT}}$, update $\mathrm{TT}(t)$ by replacing v'' with v.

The time complexity of this phase is in $O(N_{\mathrm{VT}})$. More precisely, it is linear in the size of $\mathrm{VT}(v)$ since deleted triangles already been removed from lists l'_{VT} and l''_{VT}.

Adjust partial VT relation

Eliminate redundant triangles from $\mathrm{VT}^*(v)$, by proceeding as explained in in the case of expansion.

In summary, the overall time complexity of vertex-pair contraction is in $O(N_{\mathrm{VT}} + N_{\mathrm{VV}})$.