Vol. 9, No. 2: 1–29

A Dynamic Adaptive Mesh Library Based on Stellar Operators

Luiz Velho
IMPA -- Instituto de Matematica Pura e Aplicada

Abstract. In this paper, we present a dynamic adaptive mesh library which maintains a conforming triangulation of time-varying surfaces. The user supplies an initial mesh, a surface sampling procedure, and a set of adaptation criteria. The mesh is automatically modified in order to conform to user-defined characteristics, while the surface changes over time. The mesh representation is based on a half-edge data structure without any extra storage requirements. The mesh has an underlying semiregular multiresolution structure. Furthermore, the specification of desired mesh characteristics can be based on very general adaptation rules. This scheme facilitates the development of graphics applications that manipulate triangulated surfaces. The library source code is available online.

1. Introduction

Polygonal meshes are arguably the most common representation for surfaces in geometric modeling and computer graphics.

In many applications, the surface is dynamic and changes its shape over time. This encompasses a wide range of problems, from the animation of deformable bodies to progressive multiscale for transmission and visualization. There are other applications where the surface shape itself is fixed, but the surface discretization has to change for computational reasons. This occurs, for example, in the case of Finite Element Simulations (FEM) with rigid bodies.

1086-7651/04 $0.50 per page

In the two situations described above, the application ideally should be concerned with the solution of the problem at hand, using a mesh representation only as a means to perform computations.

In practice, however, it is very common that a significant part of the effort in developing such applications goes into the task of maintaining the mesh representation. Worse yet, sometimes it is difficult to separate parts of the implementation related to the main problem domain from the mesh infrastructure, making the code less portable and error-prone.

In order to overcome these drawbacks, it would be desirable to have a mesh library that could encapsulate all the functionality for supporting a dynamic mesh representation. The implementation should be robust, computationally efficient, and economical in terms of memory space. Moreover, the Application Program Interface (API), should be simple and provide the right level of abstraction.

In this paper, we present a simple scheme for creating and maintaining a mesh representation of time-varying surfaces that addresses all the above mentioned requirements. As an additional benefit, our mesh representation has an underlying semiregular multiresolution structure which can be further exploited in various ways.

More specifically, the relevant features of this software are:

- a simple dynamic adaptive mesh library based on the half-edge, a standard topological data structure. The implementation of this new adaptive multiresolution functionality does not require any extra storage in the representation. Also, because the half-edge is widely adopted, it should be easy to incorporate the library in many applications.

- a minimal API for mesh creation and adaptation. This interface complements the traditional topological query operators and consists of only a few functions.

- a conformal mesh structure that dynamically changes its resolution based on user-defined criteria. This makes the associated adaptation capabilities very general and powerful.

- an effective mechanism for refinement and simplification of semiregular meshes that maintains a restricted multiresolution structure. This mechanism is based on the concept of a restricted binary multitriangulation and stellar theory.

Our scheme is based on edge operators. Edge operators are commonly used because of their good adaptation properties. For example, Bowden et al. [Bowden et al. 97] employ an inflating balloon model to reconstruct a surface from volumetric data. In this model the dynamic mesh is based on refinement

by edge bisection. Kobbelt et al. [Kobbelt et al. 00] propose a multiresolution shape representation based on geometry smoothing and dynamic meshes that are modified by edge collapses, edge splits and edge flips.

Existing schemes for dynamic meshes are usually developed in the context of specific applications. Unlike most previous work, our scheme is application-independent. Moreover, it relies on stellar moves on edges which have the expressive power to implement arbitrary transformations on combinatorial manifolds [Lickorish 99].

Our dynamic mesh representation has an underlying multiresolution structure. Multiresolution representations can be defined through global or local operations on a mesh [Garland 99]. In order to support adaptation, the multiresolution data structure has to be constructed using local operations. Progressive meshes [Hoppe 98] constitute one example of such data structure. Another example is the hierarchical 4-K mesh [Velho and Gomes 00]. In this kind of representation, different meshes can be dynamically extracted from the data structure. However, the local operations need to be explicitly stored. In our scheme, only the current mesh is stored in the representation. The legal moves that change the mesh resolution are implicitly defined.

To the best of our knowledge, there is no other proposal for a dynamic mesh representation that has an implicit underlying multiresolution structure and is application-independent. Nonetheless, there is a growing interest in the development of general mesh libraries [Fabri et al. 00].

The adaptation scheme proposed in this paper is orthogonal to the infrastructure existent in a general mesh library. In that sense, it complements such libraries with additional functionality that could be implemented on top of edge-based mesh representations, such as the OpenMesh [Botsch et al. 02].

The library C++ implementation with source code is available online at the address listed at the end of this paper.

2. Overview

Our library is intended for applications in which shape information is known independently of the mesh. This is particularly true when the surface shape is not static and changes dynamically over time, or has an external continuous definition. Some examples are variational modeling, multiresolution editing, surface remeshing, and visualization of parametric or implicit surfaces.

Typically in such applications, the following requirements are satisfied:

- there is a base domain where the surface geometry is defined;
- it is possible to compute samples of points on the surface.

Our library takes care of maintaining an adaptive variable resolution mesh, but storing only the current mesh and providing a simple interface. The key to achieving this goal is to impose a semiregular structure to the mesh, as will be seen in the next sections.

The user must provide the topology and geometry of an initial base polyhedron, together with a means to sample and adapt the underlying surface.

In this section, we describe the adaptive mesh library's API and give an example of using the library.

The library API, related to mesh construction and adaptation, is composed of only two functions:

- `Mesh(Surface* surf)` – This function is the mesh constructor. It takes as a parameter an object that represents the surface.
- `Mesh::adapt()` – This function adapts the mesh according to the user-defined criteria provided by `surf`.

The function `adapt` performs both refinement and simplifications of the current mesh. Two additional functions, `adapt_refine` and `adapt_simplify`, are also exposed by the library API, in case the application needs to do only refinement or simplification.

The above API requires that the application supplies four functions through the surface object: a procedure to construct the *base polyhedron*; a *sampling function* to evaluate the geometry of the surface at the vertex of the mesh; a *refinement test*; and a *simplification test* to determine if the mesh needs further subdivision or coarsening, respectively.

Therefore, the `Surface` object class has to implement the following minimum functionality:

```
class Surface {
  void base_mesh(int nv, Point pts[][], int nt, int tris[][]);
  void sample(Edge e, Vertex &v);
  float refine_test(Hedge e);
  float simplif_test(Vertex v);
}
```

In addition to the API above, the library also provides the standard operators for querying and navigating topological data structures.

The library data structure does not include any a priori geometric data. This is supplied by the application through attribute classes. The library deals only with topological issues. Therefore, the application has full control of geometry aspects through the face, edge, and vertex attribute classes and the sampling function. This mechanism makes the library more general, with the flexibility to accommodate different types of surface descriptions, such as parametric, implicit, etc.

It is worthwhile to note that the mesh has an underlying semiregular multiresolution structure. This implies that every vertex V can be labeled with a unique index (b, i, j, k), where b indicates the base triangle from which that vertex was generated nd (i, j, k) are dyadic barycentric coordinates. Other labeling schemes are possible when the surface has a particular form of parametrization. In such cases, the application incorporates this knowledge into the vertex attribute class. Parametrization is important for computing the sampling function.

3. Multiresolution and Stellar Theory

The key to our adaptive mesh representation is its underlying multiresolution structure. In this section, we review the basic multiresolution concepts and their relationship with stellar subdivision theory. For more details, see [Velho 04].

We are going to work with a mesh representation for triangulated manifold surfaces. This is not a serious restriction because every smooth surface is triangulable [Munkres 66]. Indeed, triangle meshes are a widely adopted discrete representation for surfaces.

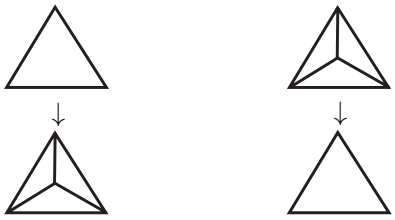
A triangle mesh is a simplicial cell complex $M = (V, E, F)$, formed by sets of simplices of dimension 0, 1, and 2. That is, vertices $v_i \in V$, edges $(v_i, v_j) \in E$, and triangles $(v_i, v_j, v_k) \in F$, respectively.

A multiresolution mesh is a monotonic sequence of simplicial complexes $H = (M_0, M_1, \ldots, M_k)$, with increasing resolution. That is, $|M_i| \leq |M_j|$ for $i < j$, where $|M|$ denotes the number of triangles of M. Furthermore, the meshes $M_i \in H$ are assumed to be equivalent triangulations of a surface S.

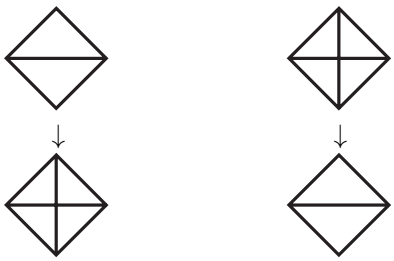
It is desirable that any two subsequent meshes M_i, M_{i+1} in the sequence H should be related by a transformation that takes one into the other. In this way, the multiresolution structure can be constructed from an initial coarse mesh by successively applying refinement transformations. If the refinement transformation has an inverse, it is also possible to build the multiresolution structure starting from a dense mesh and repeatedly making simplification transformations.

Usually, it is desirable that these mesh transformations consist of local modifications which affect only a part of the mesh. In this case, the dependency relations between any two meshes in H will be local. Based on this fact, it is possible to derive from H a dependency graph that allows the extraction of new meshes that are not present in the original sequence by piecing together independent local modifications.

From the above discussion on multiresolution structures we conclude that we need local operators to transform triangle meshes. The theory of stellar subdivision provides such operators.



(a) face split and face weld



(b) edge split and edge weld

Figure 1. Stellar subdivision operators.

The basic stellar subdivision operators for two dimensional meshes are: the *face split* and its inverse *face weld*; and *edge split* and its inverse *edge weld*. These operations are illustrated in Figure 1.

Stellar theory studies equivalences between simplicial complexes [Lickorish 99]. The main theorem of stellar theory states that stellar operators on edges can transform between equivalent combinatorial manifolds.

Stellar subdivision is relevant in the context of multiresolution because it provides the basic operators to refine and simplify a triangle mesh.

In our mesh library, face splits and welds will be used to impose a semiregular structure on the base mesh, while edge splits and welds will be used to transform the current mesh according to the underlying semiregular multiresolution structure.

4. Binary Multitriangulations

As we have seen in the previous section, there are good reasons to use stellar operations on edges. Whenever a stellar subdivision happens in an edge ε, the triangles incident in ε are split into two. Accordingly, a sequence of stellar

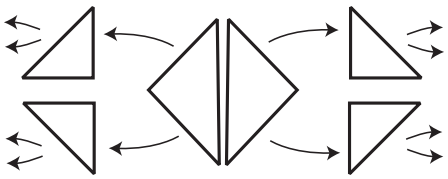


Figure 2. Binary tree hierarchies: two binary trees resulting from an edge split operation.

subdivisions on edges induces binary tree hierarchies in the simplices (see Figure 2). Binary trees are simple and often lead to efficient algorithms.

In order to construct a multiresolution structure based on edge splits we exploit the fact that stellar subdivision on edges introduce local modifications in a mesh. In fact, this operation affects only the triangles that are incident on that edge (i.e., two triangles for interior edges and one triangle for boundary edges). The submesh formed by these incident triangles is the *star* of the edge. We will also call this submesh a *basic block*. Therefore, two edge splits are independent if one edge is not contained in the star of the other (and vice-versa). It is easy to see that independent edge splits can be applied in an arbitrary order. Conversely, note that there is a dependency between a split edge and the edges on the boundary of its basic block. These dependencies induce an interleaving relationship between the binary tree hierarchies. Note that, in general, a basic block is a pair of triangles with a common internal edge, but on the border of an open mesh a basic block can be just a single triangle where the border edge is the split edge. Figure 3 illustrates the dependencies between basic blocks.

To globally change the resolution of a mesh, we can apply any set of independent local modifications. This creates a new resolution layer over the

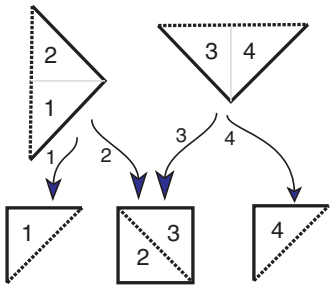


Figure 3. Dependencies and basic blocks: two different basic blocks split and their elements are grouped to form new basic blocks (split edges are indicated by dashed lines).

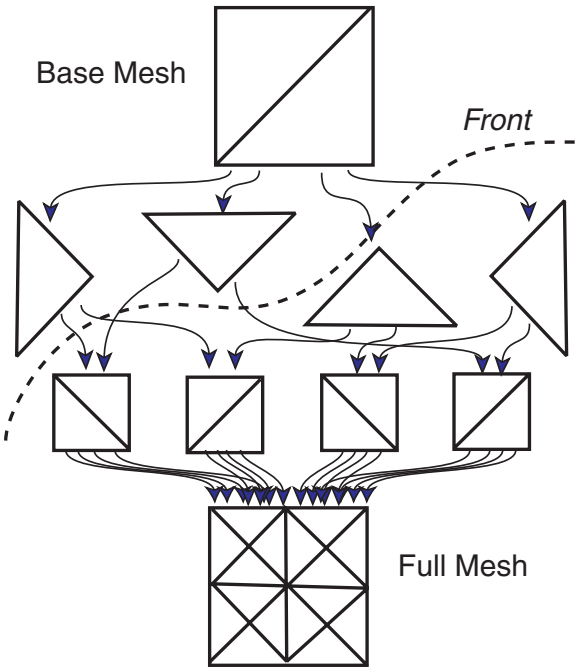


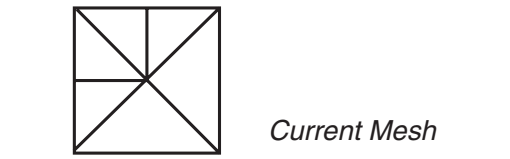
Figure 4. BMT viewed as a DAG: the dependency graph is formed by interleaved binary tree hierarchies. Each interior basic block has two incoming and four outgoing arrows, indicating respectively the triangles belonging to the block and the triangles resulting from splitting the internal edge of the block. Basic blocks on the boundary have only one incoming and two outgoing arrows.

mesh. Two subsequent resolution layers are constrained by the dependency relations of local modifications.

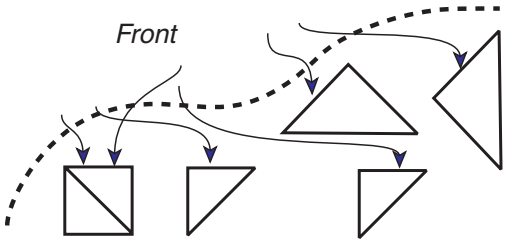
The above concepts allow us to define a *Binary Multitriangulation* (BMT). A BMT is a multiresolution structure formed by applying edge splits to an initial mesh $\overline{M}$, called *base mesh*. The final mesh resulting from this subdivision process is called *full mesh*.

The BMT can be thought as a Direct Acyclic Graph (DAG) describing all possible ways to make local changes to a mesh. In this DAG, the arrows are labeled with stellar subdivision on edges and the nodes are submeshes. From an algorithmic perspective, the key idea is to use the above mentioned binary tree hierarchies in the simplices to encode the DAG. Figure 4 illustrates this concept.

Any cut in the DAG that separates the base mesh from the full mesh represents a valid mesh, as can be seen in Figure 5; such a cut is called *front*. Note that this mechanism allows the generation of a large number of meshes,



(a)



(b)

Figure 5. Extracting a mesh from a BMT: (a) current mesh resulting from applying edge splits to the base mesh; (b) cut in the DAG of Figure 4 representing the front.

distinct from the ones defined in the multiresolution sequence. It is the flexibility in choosing how to make those cuts that warrants the expression power of a BMT.

The BMT is, in fact, a particular case of a more general variable resolution structure, called Multitriangulation [Floriani et al. 98]. Binary multitriangulations have been introduced by Velho and Gomes [Velho and Gomes 00]. We define now a more specialized version of the BMT, the regular BMT.

A *Regular Binary Multitriangulation* (RBM) is a binary multitriangulation that satisfies the following conditions:

1. the base triangulation is a *tri-quad mesh*.
2. refinement operations are only applied to interior edges of *basic blocks*.

The requirements of a regular binary multitriangulation are based on the concepts of basic-block and tri-quad mesh.

A *basic block* is a pair of triangles with a common edge, called the *internal edge* of the block. The other edges are called *external edges*. A *tri-quad mesh* is a triangulated-quadrangulation, e.g., a mesh that is the union of basic blocks. Figure 6 shows a tri-quad mesh, where the interior edges of basic blocks are rendered as dashed lines.

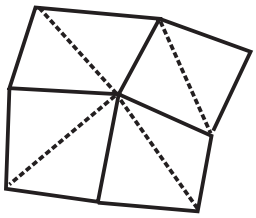


Figure 6. Example of a tri-quad mesh with four basic blocks.

Note that, when the internal edge of a basic block is subdivided by an edge split operation, it is necessary to form new basic blocks in order to regularize the mesh. This is done by subdividing adjacent blocks. In that way, the external edges of previous blocks become internal edges of new blocks. This process is called *interleaved refinement* (see Figure 7).

It is the interleaved refinement process that gives an underlying regular structure to the meshes extracted from a RBM. In addition, this process guarantees that any two adjacent triangles in a mesh will differ at most by one level of refinement. This produces what is called a *restricted structure* [Von Herzen and Barr 87].

Thus, the RBM can be viewed as a forest of interleaved quad-trees formed by basic blocks. The concept of regular binary multitriangulations was introduced by Velho and Zorin [Velho and Zorin 01], in connection with the $\sqrt{2}$ subdivision scheme.

The restricted binary multitriangulation has very special properties. It is optimal in relation to the main criteria used to measure the effectiveness of a variable resolution structure [Puppo 98]. More precisely, it achieves the highest expressive power, it has logarithm depth, and also has linear growth rate.

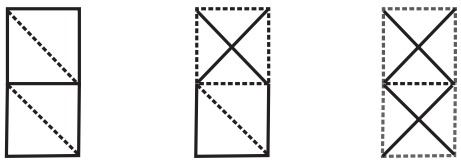


Figure 7. Interleaved refinement: (a) the mesh has two basic blocks; (b) the top basic block splits; (c) the bottom basic block has to split in order to regularize the mesh. The final mesh has seven basic blocks—one in the interior and six on the boundary.

5. Adaptive Meshes

Regular binary multitriangulations possess a very rich adaptation structure. The key element for this property is the ability to make local changes in the current mesh by moving the front around a node in the DAG.

As we mentioned before, the regularity of the RBM allows us to store only the current mesh (e.g., the front), while being able to operate as if we had the complete multiresolution structure.

The stellar operators `edge split` and `edge weld` implement just "local" transitions in the DAG, that is, if $\mathcal{T}$ and $\mathcal{T}'$ are the triangulations before and after an `edge split` is called, respectively, then $\mathcal{T}'$ is a successor of $\mathcal{T}$.

However, in order to keep the current mesh consistent it is necessary to propagate the dependencies between submeshes that are encoded in the DAG. This propagation mechanism is based on the following observation.

In a binary multitriangulation, every triangle, $\sigma \notin \overline{M} \cup \underline{M}$, not in either the base mesh $\overline{M}$ or the full mesh $\underline{M}$, has a *refining element* and an *unrefining element*.

The refining element will be a *split edge* e_s, while the unrefining element will be a *welding vertex* v_w.

This means that, before applying a stellar subdivision operation on some element of the mesh (e.g., an edge split or an edge weld), all triangles that are incident on the element (e.g., the edge to be refined or the vertex to be simplified) must have that element as the subdivision element (e.g., refining or unrefining).

This observation leads automatically to a restricted hierarchical structure. Moreover, it can be implemented using a simple recursive algorithm. It is also remarkable that this algorithm is basically the same for both refinement and simplification.

We present the pseudo C++ code of the algorithms for RBM refinement and simplification. In Algorithm 1, we describe the restricted edge refinement

Algorithm 1. (Recursive restricted edge refinement.)

```
Vertex* Mesh::refine(Edge* e)
{
  for (Iterator f = incident_faces(e))
    if (f->split_edge() != e)
      refine(f->split_edge());
  update_front_on_refine(e);
  Vertex* v = split(e);
  update_front_on_refine(v);
  return v;
}
```

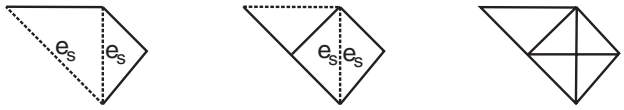


Figure 8. Recursive dependency propagation in edge refinement: the two triangles don't share a split edge because they are at different levels (a); therefore the left triangle has to be subdivided (b) before splitting the edge (c).

and in Algorithm 2 we describe the restricted vertex simplification. These two algorithms use functions for updating the fronts that are described in Algorithms 3 and 4.

Figure 8 gives an example of dependency propagation in restricted edge refinement.

In Algorithm 2, the function `max_level_neighbor(w)`, returns the vertex in the link of w with highest subdivision level. Note that the order of welding incident simplices to the welded vertex is relevant: they must be welded from the highest to the lowest level.

The method `weld_degree` returns the degree that a vertex should have for a weld to be applied. This value is operator-dependent (4 for a weld of an internal edge and 3 for a weld of a boundary edge).

Figure 9 gives an example of dependency propagation in restricted vertex simplification.

The refining and unrefining elements of the current mesh constitute valid transitions to change the front. We keep these split edges and welding vertices in two heaps corresponding to the candidate refinement and simplification operations, respectively. These heaps are ordered according to a user-defined

Algorithm 2. (Recursive restricted vertex simplification.)

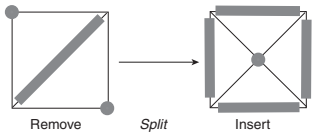
```
Edge* Mesh::simplify(Vertex* w)
{
  do {
    Vertex* v = max_level_neighbor(w);
    if (v->level() > w->level())
      simplify(v);
  } while (w->degree() != w->weld_degree());
  update_front_on_simplif(w);
  Edge* e = weld(w);
  update_front_on_simplif(e);
  return e;
}
```

(a) (b) (c)

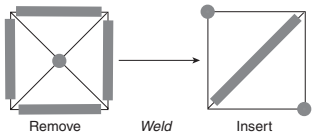
Figure 9. Recursive dependency propagation in vertex simplification: the welding vertex is not the same for all triangles in (a); therefore the four top-left triangles are simplified (b) before performing the edge weld(c).

priority, such that operations with higher priority are performed first by the global adaptation routines. The priority of local change operations is set by the functions `ref_test` and `simpl_test` that are declared in the class `Surface`. Usually, the user will define these functions based on a measure of the approximation error between the current mesh and the true surface.

The candidate transition elements of the front are maintained in refinement and simplification heaps by the routines `update_front_on_refine` and `update_front_on_simplif`. These routines are called by `refine` (Algorithm 1) and `simplify` (Algorithm 2). For each operation (refine or simplify), they are called before and after the operation to update the corresponding heap. Before the operation they remove from the heap elements that cease to be transition elements once the operation is performed. After the operation they insert in the heap elements that become transition elements when the operation is



(a) refinement update



(b) simplification update

Figure 10. Updating the front—grey elements are removed or inserted in the heap.

Algorithm 3. (Heap update on refinement.)

```
void Mesh::update_front_on_refine(Edge* e)
{
  refine_heap().remove(e);
  for (Vertex* w = link(e))
    simplif_heap().remove(w);
}

void Mesh::update_front_on_refine(Vertex* v)
{
  simplif_heap().insert(v, surf.simpl_test(v));
  for (Edge *h = link(v))
    refine_heap().insert(h, surf.ref_test(h));
}
```

done. These elements belong to the 1-neighborhood of the split edge and the welding vertex. Figure 10 shows the scheme for removing and inserting transition elements in order to update the heaps.

Algorithm 3 shows the code for updating the front during the edge split operation. Note that function overloading is used to select the method for each phase of the update process.

Algorithm 4 shows the code for updating the front during the edge weld operation. As expected, the code for updating the front for a weld operation is completely symmetric and complementary to the split operation. Here, function overloading is used as well.

The algorithms `refine` and `simplify` make nonlocal transitions in the DAG, propagating dependencies to maintain the mesh consistency. In order to perform global adaptation on a mesh, it is necessary to apply these two

Algorithm 4. (Heap update on simplification.)

```
void Mesh::update_front_on_simplif(Vertex *v)
{
  simplif_heap().remove(v);
  for (Edge *e = link(v))
    refine_heap().remove(e);
}

void Mesh::update_front_on_simplif(Edge *e)
{
  refine_heap().insert(e, surf.ref_test(e));
  for (Vertex *v = link(e))
    simplif_heap().insert(h, surf.simpl_test(v));
}
```

Algorithm 5. (Adaptive mesh refinement.)

```
void Mesh::adapt_refine(float thresh)
{
  while ((e = refine_heap().extract()) != NULL) {
    if (e.priority() < thresh) {
      refine_heap().insert(e, surf.ref_test(e));
      return;
    }
    refine(e);
  }
}
```

algorithms over the whole mesh and refine or simplify it where desired.

We remark that, as in the restricted simplification methods, the adaptation algorithm is essentially the same for refinement and simplification. The algorithm keeps scanning through the candidate transition elements for the operation (e.g., edges for refinement and vertices for simplification). For each transition element (split or weld), it asks if the transition should be made (e.g., the element priority greater than the threshold value `thresh`) and performs the operation when this condition is true.

The code for global adaptive refinement and simplification is shown in Algorithms 5 and 6, respectively.

There are a couple of observations regarding the above algorithms. First, observe that stellar operations are independent of each other because they change only the 1-neighborhood of the element. Each operation automatically maintains the restricted structure.

The user supplies the adaptation criteria through a function that is called with the transition element as its argument. This function can be very general and can be based either on local information from the mesh (e.g., queried through the element), or any other global information (e.g., can de-

Algorithm 6. (Adaptive mesh simplification.)

```
void Mesh::adapt_simplify(float thresh)
{
  while ((v = simplif_heap().extract()) != NULL) {
    if (v.priority() < t) {
      simplif_heap().insert(v, surf.simpl_test(v));
      return;
    }
    simplify(v);
  }
}
```

pend on other factors besides the mesh itself, such as camera position in a view-dependent adaptation). Also, the function is responsible for making sure that the criterium will eventually be satisfied so that the process terminates.

In order to have a complete adaptation, parts of the mesh might need to be refined, while other parts may have to be simplified. Consequently, the two functions should be called every time the conditions change, as illustrated in the code below.

```
void Mesh::adapt(float t)
{
  adapt_refine(t);
  adapt_simplify(t);
}
```

In this full adaptation case, it is mandatory that the tests for refinement and simplification are based on the same adaptation criteria and complement each other. This guarantees that the final result will be correct.

A simple example is as follows. Suppose that the criteria is to keep the mesh resolution at level `J`. Then, the test in `ref_test(e)` should be `(e.level() < J) ? 1 : -1`, while the test in `simpl_test(w)` should be `(w.level() > J + 1) ? 1 : -1`, and the threshold for the adaptation functions should be $t = 0$.

6. Data Structures

The algorithms described in the previous section are applicable to general BMTs, and also to regular BMTs.

However, in the case of general BMTs, it is necessary to explicitly store the dependency graph. This DAG essentially encodes the transition elements for refinement and simplification, as well as the geometric information associated with changes of mesh resolution. A hierarchical data structure suitable for this purposed has been introduced by Velho and Gomes [Velho and Gomes 00].

In this section, we exploit the properties of the regular BMT, in order to provide all the functionality discussed in Section 5, without the need for storing a hierarchical data structure.

The library uses a standard representation based on the half-edge data structure.

In this representation, a mesh is a collection of sets of vertices, edges, and faces. The geometric data for and the adaptation criteria is obtained through a user-supplied surface object. The front is stored in two heaps.

```
struct Mesh {
```

```
  Container<Face*>   faces;
  Container<Edge*>   edges;
  Container<Vertex*> vertices;
  Heap               refine_heap;
  Heap               simplif_heap;
  Surface            surf;
}
```

A triangular face is a loop of three oriented half-edges.

```
struct Face {
  Half_Edge* he_ref;
}
```

An edge stores two half-edges, one for each orientation.

```
struct Edge {
  Half_Edge  he[2];
}
```

The half-edge contains pointers to its origin vertex, the next half-edge in the loop, the face corresponding to that loop, and its parent edge.

```
struct Half_Edge {
  Vertex*     org_ref;
  Half_Edge*  nxt_ref;
  Face*       f_ref;
  Edge*       e_ref;
}
```

The vertex stores its subdivision level, and a pointer to one of the incident edges in its star.

```
struct Vertex {
  Half_Edge* star_ref;
  int        level;
}
```

In order to refine and simplify the current mesh, it is necessary to know the split edge and weld vertex of a triangle. We want to obtain this information without having to store the data explicitly. This is possible because of the regular structure of the RBM. Note that in an RBM, the split edge is always opposite to the weld vertex, as shown in Figure 11. Therefore, we establish a convention that the split edge and the weld vertex are numbered as the first simplices of dimension 1 and 0 of a triangle, e.g., $e_0 \in (e_0, e_1, e_2)$ and $v_0 \in (v_0, v_1, v_2)$. This numbering is consistent with the face operator definition for abstract simplicial complexes [May 67].

The scheme is automatically maintained by our library's implementation of the stellar operators `split` and `weld`. Figure 11 shows the zero-placement

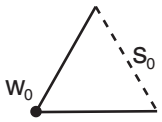


Figure 11. Zero-placement scheme for subdivision elements.

scheme for subdivision elements.

With the zero-placement scheme above we have all the information necessary to implement the split operator. However, the weld operator requires one more piece of information. It is given a weld vertex w and must transform the star of w into two triangles. Thus, it needs to know which vertices from the link of w should be connected to create the new internal edge of the block. We encode this information, using the convention that the origin vertex of the welded edge is the same as the origin of the half-edge indicating the star of w. This scheme is shown in Figure 12.

The data structure encodes the relevant information for the implementation of stellar operators as follows:

- each element has sufficient information to recover its incident elements and its star;
- the split edge of a triangular face is given by the 0th half-edge of the loop.
- the welded vertex of a triangular face is given by the 0th vertex of the loop. Note that this is a consequence of the previous item.
- the star of a vertex is indicated by a pointer to one of its incident half-edges. This half-edge also indicates the origin of the welded edge.
- an edge is on the boundary if its mate half-edge points to a null face. Analogously for a boundary vertex, we have that the first incident half-edge is a boundary edge.

The mesh representation also allows all the standard topological queries for navigating a combinatorial manifold structure. Most operators are easily

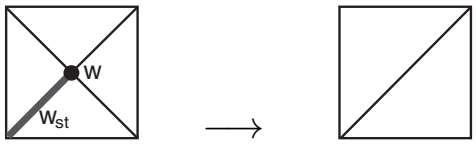


Figure 12. Scheme for encoding the weld of an edge.

inferred by inspection of the data structures. Below we list the methods associated with each data item.

The face methods are:

- `Hedge* Face::hedge(int i)` – returns the ith half-edge;
- `Vertex* Face::vertex(int i)` – returns the ith vertex;
- `Hedge* Face::split_hedge()` – returns the split half-edge;
- `Vertex* Face::weld_vertex()` – returns the welding vertex;
- `boolean Face::is_inbase()` – returns true if the face is part of the base mesh.

The edge methods are:

- `Hedge Edge::hedge(int i)` – returns the ith half-edge;
- `boolean Edge::is_boundary()` – returns true if on boundary;
- `boolean Edge::is_split()` – returns true if it is a split edge.

The half-edge methods are:

- `Vertex* Hedge::org()` – returns the origin vertex;
- `Vertex* Hedge::dst()` – returns the destination vertex;
- `Face* Hedge::face()` – returns the incident face;
- `Edge* Hedge::edge()` – returns the parent edge;
- `Hedge* Hedge::sym()` – returns the mate half-edge;
- `Hedge* Hedge::prev()` – returns the previous half-edge;
- `Hedge* Hedge::next()` – returns the next half-edge.

The vertex methods are:

- `Hedge* Vertex::star_first()` – first incident half-edge;
- `Hedge* Vertex::star_next(Hedge *e)` – iterates to get the next half-edge incident on the vertex;
- `Point* Vertex::level()` – returns the level of the vertex.

In addition to the mesh adaptation functions,the library also provides iterators for accessing the mesh elements:

- `Iterator Mesh::face_iter()` – returns face container iterator;
- `Iterator Mesh::edge_iter()` – returns edge container iterator;
- `Iterator Mesh::vertex_iter()` – returns vertex container iterator.

The data structure described above contains only topological information. In our $C++$ implementation, the classes for topological elements (vertex, edge, face) are the bases for classes of geometrical elements. This mechanism enables new attributes to be added to any element. One can, for instance, add a geometric position to each vertex, or a surface normal to each face, or a scalar value to each edge. A similar technique based on attribute classes is used in Botsch et al. [Botsch et al. 02].

In summary, our mesh representation uses a standard topological data structure, and is able to implicitly encode all information necessary for maintaining a dynamic variable-resolution mesh with the underlying structure of a RBM.

7. Building the Base Mesh

The mesh constructor takes as a parameter an object of the class `Surface`. As mentioned before, the surface object encapsulates the global geometric and topological information used by the mesh library, including the definition of the base domain and sampling the surface. Algorithm 7 shows the pseudocode for the mesh constructor.

When a new mesh is instantiated, the constructor calls `Surface::base_mesh` to initialize the data structures. This function must specify the geometry and topology of the base domain through arrays of points and indexed triangles, respectively.

Algorithm 7. (Mesh constructor.)

```
Mesh(Surface s)
{
  surf = s;
  s.base_mesh(nv, verts, nf, faces);
  set_mesh(nv, verts, nf, faces);
  if (!is_triquad_mesh())
    make_triquad();
  init_heaps();
}
```

Algorithm 8. (Generation of a tri-quad mesh.)

```
void Mesh::make_triquad(void)
{
  priority_queue<node,vector<node>,less<node>> q;
  for (e = edges_begin(); e != edges_end(); e++) {
    e->set_mark(false);
    q.push(node(e));
  }
  while ( !q.empty() ) {
    node n = q.top(); q.pop();
    if ( !n.edge->is_marked() ) {
      mark_link(n.edge);
      split(n.edge);
    }
  }
  for (f = faces_begin(); f != faces_end(); f++)
    if (f->level() == 0)
      split(f);
}
```

From this description, an initial mesh structure is generated. Since the base mesh must be a triangulated quadrangulation, the mesh constructor tests if the user-supplied description has this property. If this is not the case, then the constructor calls a procedure to transform the base mesh into a triangulated quadrangulation. After that, the heaps are initialized with the transition elements.

We note that it is always possible to transform an arbitrary triangular mesh into a triangulated quadrangulation. An algorithm to create tri-quad meshes is given in [Velho and Zorin 01]. We include it here for completeness.

The tri-quad mesh generation algorithm shown in Algorithm 8 works as follows: the mesh is decomposed into clusters of triangle pairs and isolated single triangles. The boundaries of these clusters will form the internal edges of basic blocks in the tri-quad mesh. This is done by subdividing triangle pairs with the edge split operator and single triangles with the face split operator. Note that, in this algorithm, triangle pairs are created based on longest edge clustering [Rivara 84].

Obviously, when the input mesh already possesses a quadrilateral structure, it is not necessary to apply the tri-quad mesh algorithm. This is automatically detected by the mesh constructor.

8. Examples of Applications

We have implemented the dynamic adaptive mesh library on several platforms using the C language and the C++ language for both Linux and Windows operating systems. We also implemented the library in Java.

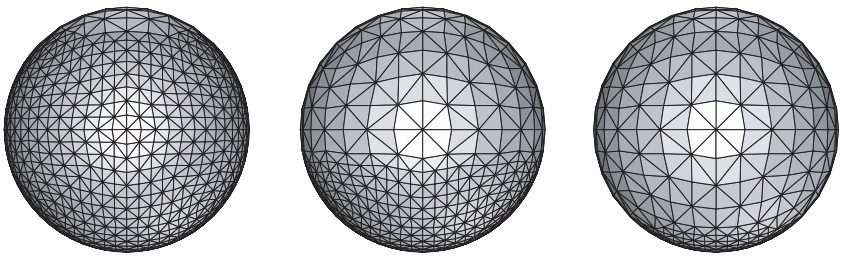
Algorithm 9. (Application main loop.)

```
Mesh m = mesh(mysurf);
while ( do_processing() ) {
  update_mesh(m);
  m.adapt();
}
```

In this section, we give examples of different applications developed with the library on these platforms. All applications have a similar structure. They create a mesh and go into a loop where the surface is modified and the mesh changes its resolution adaptively, as shown in Algorithm 9. The main difference between the applications is how the surface is specified and which adaptation criteria is employed.

8.1. Basic Surface Subdivision

In this example the surface is a parametrically defined sphere. The adaptation criteria is region-based. The user interactively controls the height of a horizontal plane that divides the sphere in two regions. Above the plane the mesh resolution is set to $N + 1$ and below to $N - 1$, where N is a reference resolution level. This program was implemented in the C language as a module of Geomview [Levy et al. 92]. Figure 13(a)–(c) shows three snapshots of an interactive session.



(a) (b) (c)

Figure 13. Geomview module: region-based adaptation. The user controls a horizontal plane dividing the mesh in two regions with resolution $N - 1$ and $N + 1$.

8.2. Image Iso-Contour Processing

In this application we are interested in the extraction and transmission of iso-valued contours of an image for analysis of medical data. The image domain is decomposed using an adaptive mesh. The mesh is subdivided around the tubular neighborhood of the iso-curve and the submesh that intersects the curve is used for processing. The application was developed in C++ with the wxWindows toolkit [Smart 97].

Figure 14 describes several stages of the process. Figure 14(a) shows the input image: CT scan of a human head. Figure 14(b) shows the contour corresponding to a user-selected iso-value. Figure 14(c) shows the mesh adapted to the iso-curve. The mesh is refined up to the maximum resolution of the image data at the iso-contour. Note that the resolution increases only near the curve. Figure 14(d) shows the intersecting submesh that is used for processing and transmission of the iso-curve.

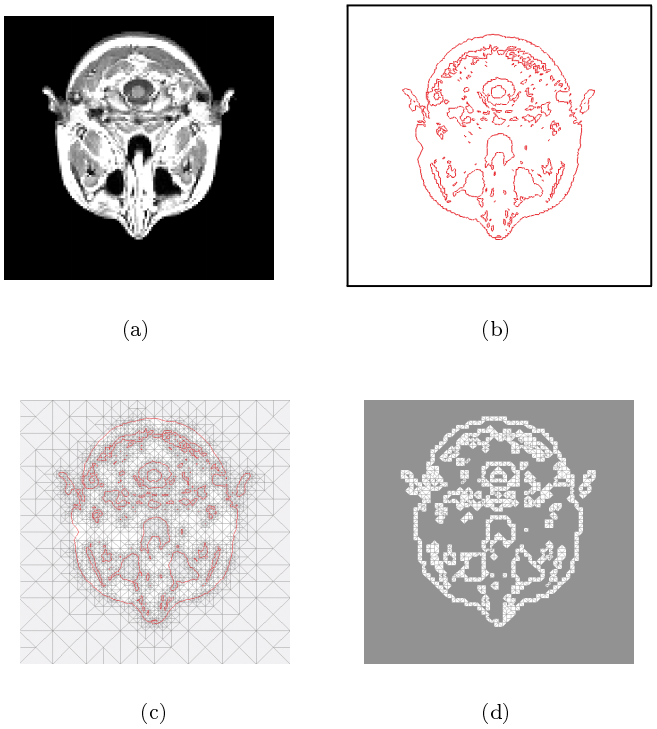


Figure 14. Image processing application: the adaptive mesh is used for extraction and transmission of iso-valued contours in medical data.

8.3. Mesh Generation for Finite Element Analysis (FEM)

Geological models often contain multiregions, complex borders, faults, holes, and other aggravating characteristics that make the life of the mesh tessellators very hard. These models often run through numerical simulations that are sensitive to small angles and negative areas. The goal of the application is to create a mesh that captures well the boundaries between different regions of a geological model and contains triangles with a good aspect ratio that are suitable for FEM analysis. For this purpose we use a combination of adaptive subdivision and physically based deformations. Subdivision refines the domain around boundaries, while deformation moves vertices to the boundaries and, at the same time, maintains the aspect ratios of triangles [Marroquim et al. 04]. The application was developed using C++ and the GLUI toolkit [Rademacher 98].

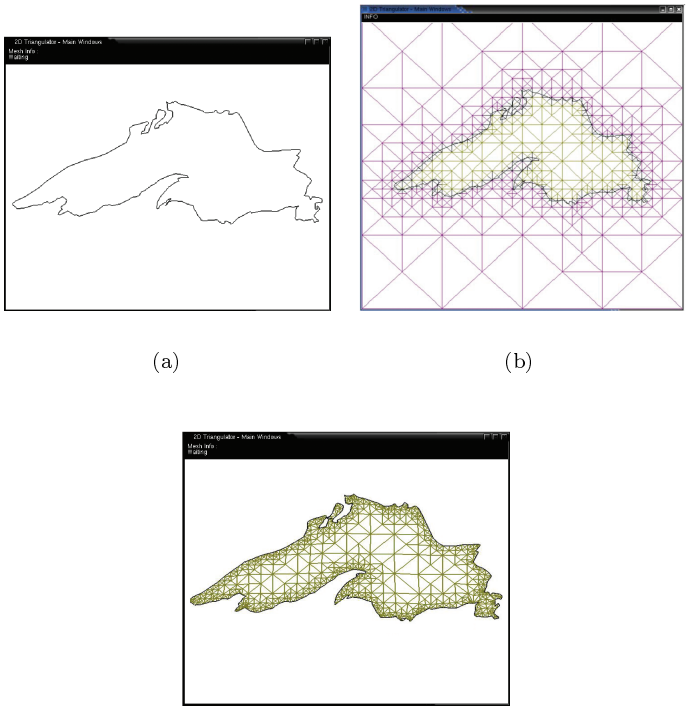
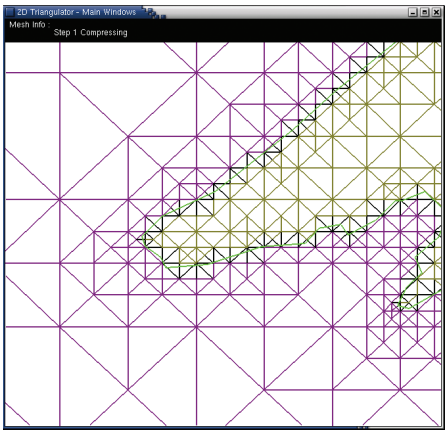
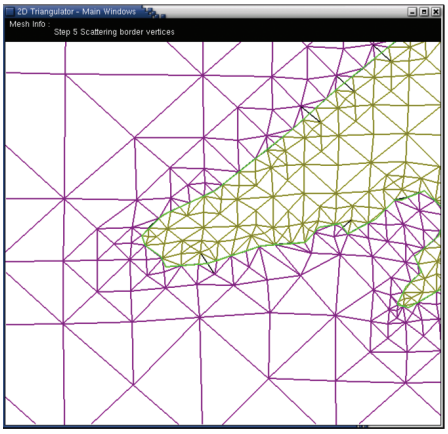


Figure 15. Triangulation of Lake Superior in Canada.

Figures 15 and 16 give some results of this application. Figure 15(a) shows Lake Superior in Canada, a classic model in the meshing area. Figure 15(b) shows the adaptive mesh without deformation. Figure 15(c) shows the final triangulation of the interior region. The mesh has 1360 triangles and all angles are between [6.070, 164.6]. Figure 16 shows an enlarged view of the undeformed and deformed meshes.



(a)

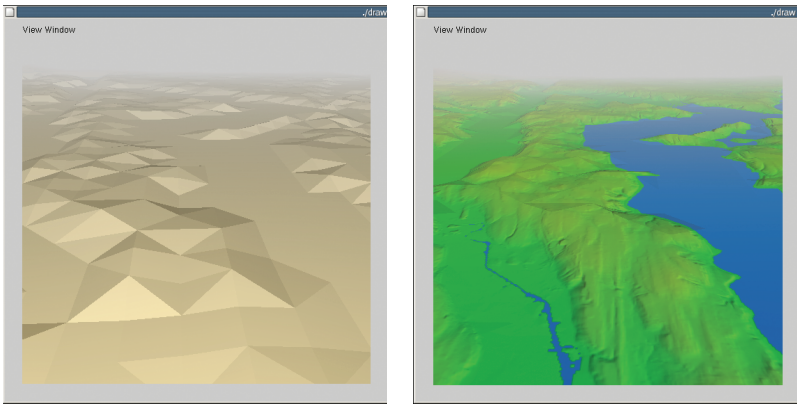


(b)

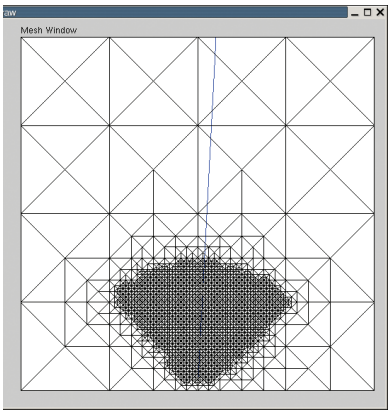
Figure 16. Detail of the undeformed and deformed meshes.

8.4. Terrain Visualization

This application was developed in C++ using OpenGL. Here, the mesh adaptation criteria is view-dependent. Figures 17(a) and (b) show, respectively, a flat-shaded and a texture-mapped rendering of the terrain. Figure 17(c) shows a top view of the mesh in wire-frame. Note that the mesh is only refined inside the view-frustrum.



(a) (b)



(c)

Figure 17. View-dependent terrain visualization.

8.5. Deformation-Based Modeling and Animation

This application was developed in Java based on Ken Perlin's Clay Modeler [Perlin 03]. Local warpings are applied to the three-dimensional ambient space in which the surface is contained. As the surface is deformed under the action of these operations, the mesh is adapted based on the surface curvature. Figure 18 shows an animated face model constructed using this technique. The initial surface is a single sphere, procedurally modified by multiple space-varying displacements. About 30 such displacements are applied to define all the bones and muscles. Expressions and emotions are animated by modifying displacements via time-varying noise functions.

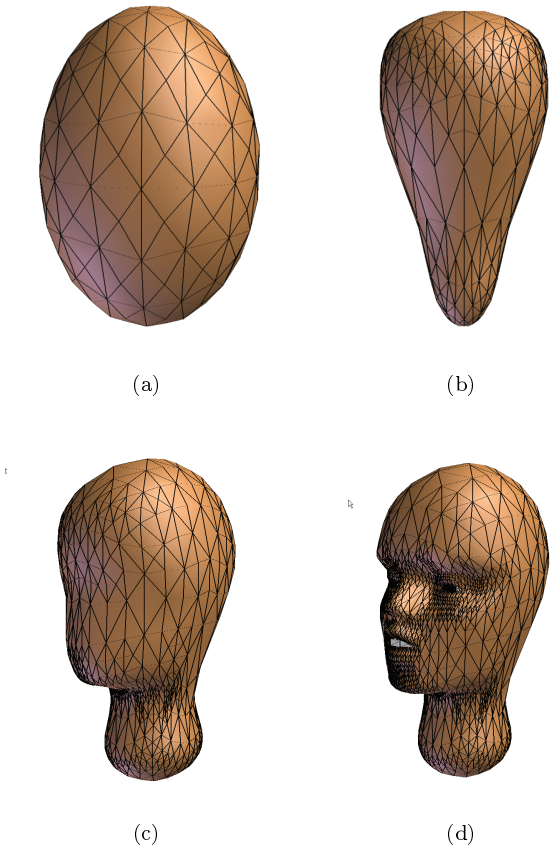


Figure 18. Animated face model using space-based deformations.

Acknowledgments. This research was partially supported by research grants from the Brazilian Council for Scientific and Technological Development (CNPq) and Rio de Janeiro Research Foundation (FAPERJ).

We would like to thank Vinicius Mello and Rubens Levy for helping in the development of the library. Many thanks also to Paulo Roma, Ricardo Marroquim, Thomas Lewiner, Ken Perlin, and Aloysio Paiva who collaborated in the development of the applications.

References

[Botsch et al. 02] M. Botsch, S. Steinberg, S. Bischoff, and L. Kobbelt. "Openmesh - A Generic and Efficient Polygon Mesh Data Structure." In *OpenSG PLUS Symposium*, pp. [XX–XX]. [CITY]: [PUB], 2002.

[Bowden et al. 97] R. Bowden, T. Mitchell, and M. Sahardi. "Real-Time Dynamic Deformable Meshes for Volumetric Segmentation and Visualization." In *Proc. BMVC 11*, pp. 310–1319. [CITY]: [PUB], 1997.

[Fabri et al. 00] A. Fabri, G. -J. Giezeman, L. Kettner, S. Schirra, and S. Schonherr. "On the Design of CGAL–a Computational Geometry Algorithms Library." *SP&E 30* 11 (2000), 1167–1202.

[Floriani et al. 98] L. D. Floriani, P. Magillo, and E. Puppo. "Efficient Implementation of Multitriangulations." In *IEEE Visualization '98*, edited by D. Ebert, H. Hagen, and H. Rushmeier, pp. 43–50. [CITY]: [PUB], 1998.

[Garland 99] M. Garland. "Multiresolution Modeling: Survey & Future Opportunities." In *Eurographics '99, State of the Art Report (STAR)*, edited by [EDITOR], pp. [XX–XX]. [CITY]: [PUB], 1999.

[Hoppe 98] H. Hoppe. "Efficient Implementation of Progressive Meshes." *Computers & Graphics* 22:1 (1998), 27–36.

[Kobbelt et al. 00] L. P. Kobbelt, T. Bareuther, and H. -P. Seidel. "Multiresolution Shape Deformations for Meshes with Dynamic Vertex Connectivity." *Computer Graphics Forum* 19:3 (2000), [XX–XX].

[Levy et al. 92] S. Levy, T. Munzner, and M. Phillips. "Geomview." Software written at the Geometry Center, University of Minnesota. Available from World Wide Web (http://www.geom.umn.edu/software/), 1992.

[Lickorish 99] W. B. R. Lickorish. "Simplicial Moves on Complexes and Manifolds." In *Proceedings of the Kirbyfest*, Vol. 2, pp. 299–320. [CITY]: [PUB], 1999.

[Marroquim et al. 04] R. Marroquim, P. R. Cavalcanti, L. Velho, and C. Esperanca. "Multi-Resolution Triangulations with Adaptation to the Domain Based on Physical Compression." Submitted, 2004.

[May 67] J. P. May. *Simplicial Objects in Algebraic Topology*, Vol. 11. Princeton, NJ: D. Van Nostrand Company, Inc., 1967.

[Munkres 66] J. Munkres. *Elementary Differential Topology*. Princeton, NJ: Princeton University Press, 1966.

[Perlin 03] K. Perlin. "Clay Becomes Flesh." Available from World Wide Web (http://www.mrl.nyu.edu/perlin/experiments/head), 2003.

[Puppo 98] E. Puppo. "Variable Resolution Triangulations." *Computational Geometry Theory and Applications* 11:34 (1998), 219–238.

[Rademacher 98] P. Rademacher. "Glui User Interface Library." Available from World Wide Web (http://www.cs.unc.edu/ rademach/glui/), 1998.

[Rivara 84] M. C. Rivara. "Mesh Refinement Processes Based on the Generalized Bisection of Simplices." *SIAM J. Numer. Anal.* 21:3 (1984), 604–613.

[Smart 97] J. Smart. "wxwindows: Cross-Platform Native UI Framework." Available from World Wide Web (http://www.wxwindows.org/), 1997.

[Velho and Gomes 00] L. Velho and J. Gomes. "Variable Resolution 4-K Meshes: Concepts and Applications." *Computer Graphics Forum* 19 (2000), 195–212.

[Velho and Zorin 01] L. Velho and D. Zorin. "4-8 Subdivision." *Computer-Aided Geometric Design* 18:5 (2001), 397–427 (special issue on subdivision techniques).

[Velho 04] L. Velho. "Dynamic Adaptive Meshes and Stellar Theory." Technical Report TR-2004-01, IMPA - Laboratorio VISGRAF, 2004.

[Von Herzen and Barr 87] B. Von Herzen and A. H. Barr. "Accurate Triangulations of Deformed, Intersecting Surfaces." *Proc. SIGGRAPH '87, Computer Graphics Proceedings* 21 (1987), 103–110.

Web Information

The software is available at http://www.acm.org/jgt/Velho04. This distribution contains the C++ source code of the library implementation and additional material, such as a demo program, documentation, links, and images.

Luiz Velho, IMPA – Instituto de Matematica Pura e Aplicada, Estrada Dona Castorina, 110 sala 326, Rio de Janeiro, RJ, 22460-320, Brazil (lvelho@impa.br)

Received July 3, 2003; accepted in revised form February 2, 2004.