Dynamic View-Dependent Multiresolution on a Client-Server Architecture

Leila De Floriani, Paola Magillo, Franco Morando and Enrico Puppo

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

Abstract

We consider the problem of transmitting huge triangle meshes in the context of a Web-like client-server architecture. Approximations of the original mesh are transmitted by applying selective refinement. A multiresolution geometric model is maintained by the server. A client may query the server for a mesh at an arbitrary, continuously variable, level of detail. The client makes repeated queries over time with different query parameters. The server answers to queries by traversing the multiresolution model and transmitting updates to the client, which uses them to progressively modify a current mesh.

We study this problem in the context of a vertex-based multiresolution model, which is a special instance of the Multi-Triangulation [Pup96, DFPM97], based on vertex insertion and removal. We define a compact data structure for such a model which exploits the specific update rule. We propose a dynamic algorithm for selective refinement and we discuss in detail its implementation as a client-server application. In order to reduce memory requirements and channel traffic, we develop a compressed representation which allows us to express mesh updates with a code of small size. We also address client caching to further limit bandwidth occupancy. Experimental results show that the Multi-Triangulation can be a key Web technology for triangle mesh manipulation.

1 Introduction

Solid objects in CAD are often represented by boundary models. A surface is given as a collection of patches, each patch being the instantiation of some parametric surface from a rather small family (e.g., NURBS). However, boundary models are often converted into raw triangle meshes for both technical, pragmatic and commercial reasons. This is accomplished by discretizing each patch of the CAD model into a set of triangles. Discretization is necessary, e.g., for display, simulation and analysis purposes. In many application domains, the result of this conversion is usually a very large mesh. Examples are large-scale design reviews, architectural demos, virtual environments, object reconstruction from 3D range devices, and terrains in geographic information systems.

The resolution, or level of detail (LOD), of a mesh is proportional to the density of its triangles, thus to their number. Meshes at very high resolution are widely available, but their huge size requires storage and computational power that is available only on powerful and expensive workstations. Even if powerful machines are becoming rather cheap and available, the bandwidth necessary to transmit such meshes is still hard to find. Therefore, Web 3D applications still deliver very simple and sketched virtual environments, leaving the problem of supporting CAD teamwork in the a Web environment unsolved.

However, for most operations we do not need the full detail in each part of the scene, but only in some interesting areas. *Selectively refined* meshes are meshes whose level of detail is variable in space according to some criterion defined by the user. They reduce memory and computational costs and can be obtained by a process of on-line simplification which removes details from a large dense mesh. This is the basic motivation behind the development of many algorithms for triangle mesh simplification. However, mesh simplification is too time consuming to be efficiently performed on line.

A more recent approach consists of decoupling the simplification phase from the selective refinement phase. This has led to the development of multiresolution models: they are essentially a compact way of encoding the simplification steps as a partial order, from which a virtually continuous set of meshes at different LODs can be extracted, based on efficient graph traversal strategies. The criteria determining the desired LOD for a selectively refined mesh typically vary with time and thus dynamic algorithms are required which can efficiently update a previously extracted mesh without necessarily performing a complete traversal of the multiresolution data structure.

Many authors have developed multiresolution structures for regular or irregular triangle meshes (see [HG97, PS97, Gar99] for recent surveys). However, all existing proposals for multiresolution data structures assume that the process handling the multiresolution model and the one which uses the extracted meshes reside on the same machine.

In this paper, we consider instead a client-server architecture in which the server stores a multiresolution data structure and contains some mechanism for extracting selectively refined meshes from it. A client sends requests to the server by specifying the required LOD (which usually may vary in different parts of the mesh) and the maximum size of a mesh that can be managed locally. The server selectively refines a mesh according to the client requests, and transmits it progressively to the client. The client processes such mesh locally, and possibly generates another request when the LOD of the current mesh becomes not satisfactory. Our assumption is based on the consideration that clients have limited on-board memory and processing power, so they can neither store a multiresolution data structure, nor receive a mesh at full resolution and run a simplification algorithm on it.

The main issue here is reducing communication time and, thus, the amount of information sent from the server to the client to describe the extracted mesh. Also, we need a compact data structure for a multiresolution model to handle huge meshes on the server, and an efficient selective refinement algorithm capable of dynamically modifying previously extracted meshes and operating in a client-server environment.

We develop here an instance of the Multi-Triangulation as defined in [Pup96, DFPM97], that we call a *Vertex-based Multi-Triangulation (MT)*, since it is built based on a vertex decimation strategy. Vertex decimation is a very popular technique used in simplification which allows preserving the topology of the mesh and the shape of the triangles (which is important when simplifying, for instance, meshes generated according to the Delaunay criterion). A Vertex-based MT is represented as a DAG in which the root provides a base mesh, while each other node describes a refinement update in the form of vertex insertion.

The major contributions of the paper are:

- An algorithm for selective refinement on the Vertex-based MT, which is dynamic, thus allowing incremental modifications on a previously extracted mesh, and generates only meshes of controlled size;
- An efficient implementation of such algorithm in a client-server environment;
- A compressed representation for mesh updates;

- A compact representation for the MT, based on the above compressed description for its nodes;
- A collection of mechanisms for handling communication between data structures on server and on the client;
- A client-caching mechanism to avoid retransmitting all details at each update.

Experimental results show that a relatively small cache severely reduces bandwidth occupancy. Yet without cache, the server answer to a client query can be coded with just a few Kbytes.

The remainder of the paper is organized as follows. In Section 2, we review previous related literature. In Section 3, we briefly describe the Multi-Triangulation, and give a formulation for the problem of selective refinement. In Section 4, we propose a dynamic algorithm for selective refinement on a Vertex MT. In Section 5, we discuss the realization of the algorithm in a client-server environment. In Section 6, we describe the data structures necessary to support the algorithm both on the server and on the client, and we provide the details of the implementation. In Section 7, we evaluate the proposed method through an experimental simulation. Finally, in Section 8, we provide some concluding remarks.

2 Related Work

A lot of work has been done in the last few years on simplification of triangle meshes, and a number of multiresolution data structures have been proposed. The interested reader can refer, e.g., to [CMS98, Gar99, HG97, PS97] for recent surveys. In this Section, we review just the papers that consider a client-server environment.

In such contexts, there are two basic issues: reducing the amount of information sent from server to client to describe the extracted mesh, and supporting the client-server interaction for the selection of the level of refinement.

The first issue has been addressed in the literature as the problem of *progressive transmission* of a mesh. The basic idea is essentially the 3D analogous to that underlying interlaced formats for images (e.g., the interlaced GIF format). A coarse mesh is sent initially, followed by a sequence of details at progressively increasing resolution. Details in the sequence are of different types and are encoded in different ways depending on the specific method.

In *Progressive Meshes* developed by Hoppe [Hop96] details consist of vertex splits. A *vertex split* is an operation that expands a vertex v of the current mesh into an edge $e = < v, v' >$ and two edges e_1 and e_2 incident at v into two triangles. Each vertex split in the sequence admits a compact encoding, which consists of the new vertex v', a reference to the old vertex v, and a code specifying the position of e_1 and e_2 around v [Hop98]. Selective refinement is obtained by sending the base mesh, followed by a subsequence of vertex splits, obtained by skipping some splits of the initial sequence where detail is not requested [Hop97, XESV97].

Taubin et al. [Tau98, GTLH99] also use a sequence of vertex splits, but they organize them into *levels* in a data structure, called a *Progressive Forest Split*, and consider the result of all splits of one level as an atomic modification to the mesh. Such modifications are transmitted in batches, which also admit a compact encoding. This format has been adopted in the MPEG4 standard as the hierarchical encoding structure of triangle meshes for progressive transmission [Koe99]. In [GTLH99] Gueziec et al. also use such technique to enrich VRML with the capability of progressively displaying a 3D model before having downloaded it completely.

The work by Snoeyink and Van Kreveld [SvK97] is limited to meshes describing terrains based on Delaunay triangulations. Here, a detail consists of inserting a vertex in the current triangulation, and it is encoded by providing just the vertex since the updating rule is implicit in the Delaunay criterion. They iteratively send batches of points, each of which forms a *layer* and increases resolution uniformly on the whole mesh. All the points in a layer must be inserted before inserting a point of the next layer. Encoding is extremely compact, because it just requires the list of vertices, plus stop bits to separate batches. On the other hand, this method offers limited possibility of adapting LOD to user requirements, it is applicable only to some special cases, and it requires numerical computation at the client, which can slow down mesh reconstruction considerably.

Note that, when meshes are handled in an environment, such as a 3D Web browser, which places the user *within* the downloading shape, optimal shape downloading should be both *progressive* and *selective*: it must be progressive to render details as they comes; and it must be selective to download most significant details first. In this scenario, typically consecutive requests of a client will produce meshes that are very similar. For instance, high resolution is desired just in proximity of an area of interest (e.g., in front of the viewpoint for landscape visualization), and such an area changes gradually in time. In this case, instead of sending the whole mesh at each extraction, just the "differences" between the current and the new mesh can be transmitted.

Techniques described above support just the incremental addition of details to a mesh, but not their deletion (when these are not needed any more). Moreover, only few of them support selective refinement. Another limitation of these approaches is in assuming a monodirectional communication from server to client, in which the server has the control on generation of LODs, and the client simply receives the data. The only decision left to the client is that of truncating the communication as soon as the resolution of the current mesh is sufficient for its needs. However, in selective refinement, the client must have the active role of deciding the LODs that it wants to receive.

The need for a client-driven selection of the LOD, and the possibility of dynamically adding and removing details of a mesh, has been mentioned in some papers (e.g., [Tau98, GTLH99]). However, to the best of our knowledge, at the state of the art there are no methods capable of supporting such features.

3 Vertex-based Multi-Triangulation

The *Multi-Triangulation* (MT) is a general multiresolution model based on local updates that we introduced in [Pup96, DFPM97]. In this section, we briefly review the model and we describe a special version of it, which is based on vertex insertion and deletion.

3.1 Preliminaries

Consider a process that starts with a coarse mesh and progressively refines it by performing a sequence of local updates. A *local update* is an operation that replaces a set of triangles with another, more numerous, set of triangles covering the same region. A sequence of local updates can be obtained, for instance, by reversing a process of mesh decimation or iterative edge collapse, such as those adopted in popular mesh simplification algorithms [CCMS97, GH97, Hop96, SZL92]. Although the MT can be defined generically for any arbitrary type of update, in this paper we will consider only a specific type, i.e., *vertex insertion*, and its reverse *vertex removal* (see Figure 1).

Under this assumption, a local update u is in one-to-one correspondence with a point v_u and it is denoted with a pair $u \equiv (u^-, u^+)$, where u^+ is an operation that refines a mesh by inserting v_u as a new vertex and modifying the mesh according to some predefined rule; and u^- is an operation that coarsens a mesh by removing vertex v_u from it, and retriangulating the corresponding hole accordingly. Alternatively, u^+ and u^- can be seen as sets of triangles: triangles of u^+ appear when we insert v_u and disappear when we remove v_u; conversely, triangles of u^- appear when we remove

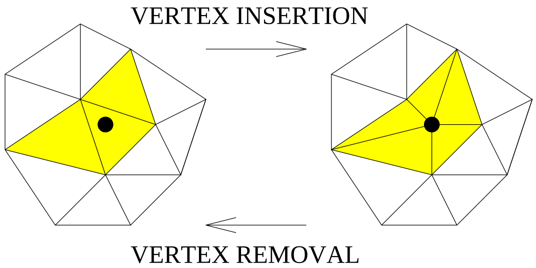

Figure 1: Modification of a triangle mesh through vertex insertion and vertex removal.

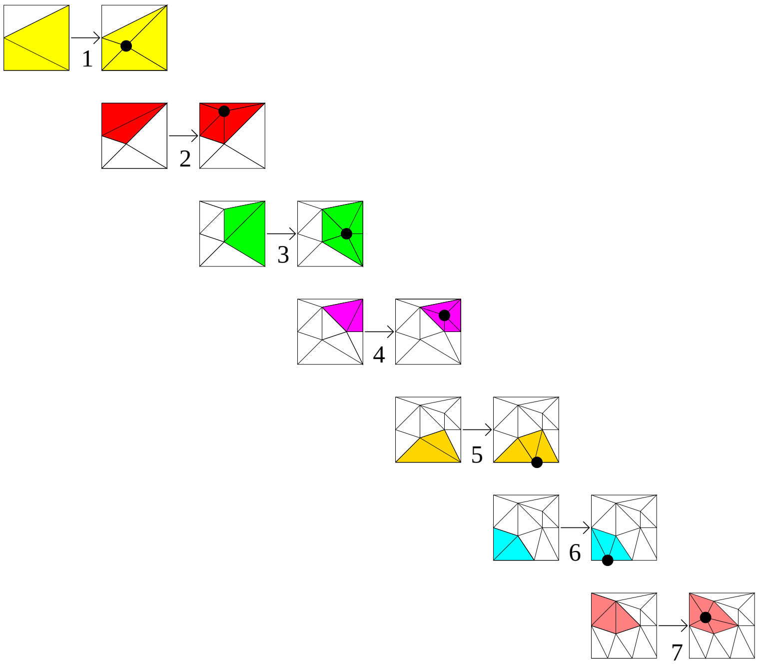

Figure 2: A sequence of vertex insertions. The mesh portion affected by each update is shown in color.

v_u and disappear when we insert v_u. Therefore, we always have $\#u^+ = \#u^- + 2$, where symbol $\#$ denotes the number of triangles in a set. Both sets u^+ and u^- are triangulations filling a simple polygon, and we call each of these two sets of triangles a *patch*. In the following, we will refer to u^+ and u^- as either operations or sets of triangles, interchangeably.

Figure 2 shows a sequence of vertex insertions. Some updates in a sequence affect separate portions of the mesh and can be performed in either order, and thus they are *mutually independent*. Other updates *depend on each other* because some triangles introduced by one update are removed by the other update. For instance, in Figure 2, updates 4 and 5 are mutually independent, and they both depend on update 3. Thus, update 4 or 5 can only be performed after 3 has been performed, but 4 can be performed without 5, or 5 without 4.

3.2 The Multi-Triangulation

A Multi-Triangulation (MT) abstracts from the *totally ordered* sequence of updates: it encodes a *partial order* describing the *mutual dependencies* between updates as a directed acyclic graph (DAG). Figure 3 shows the MT corresponding to the sequence of updates of Figure 2.

Nodes of the DAG represent updates, while arcs represent dependency relations between updates: an arc (u_1, u_2) exists whenever a non-empty subset of the triangles introduced by u_1 are removed by u_2. The creation of the initial coarse triangle mesh is represented by a special node, which is the unique root of the DAG. In the following, we refer to a node and to the update defining it interchangeably, by using the same symbol u.

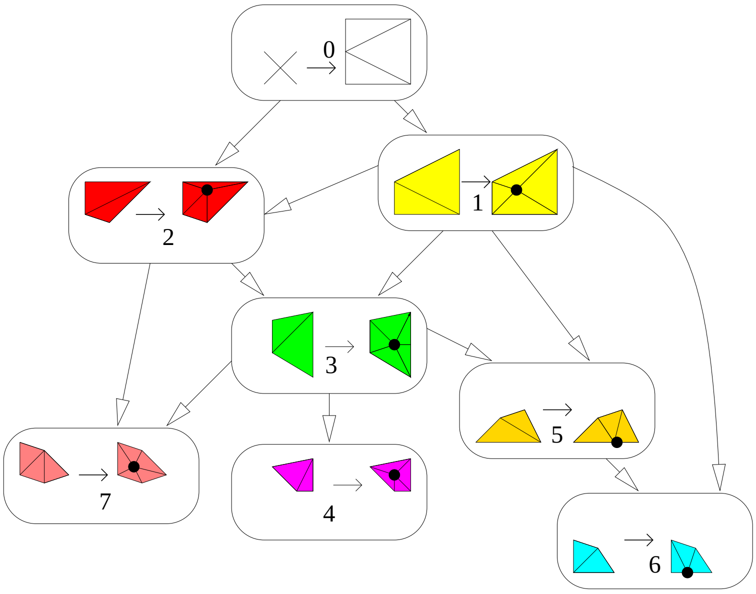

Figure 3: The Multi-Triangulation corresponding to the sequence of vertex insertions of Figure 2.

3.3 Triangle Meshes from an MT

Many different meshes at different levels of detail can be obtained by performing different subsets of updates that appear in the MT. We say that a non-empty subset S of updates of an MT is *consistent with the partial order* (or, simply, *consistent*) if, for every node u belonging to S, all the parents of u are also in S. Updates of a consistent subset can be applied to the initial mesh in any total order that extends the partial order encoded in the DAG, to produce a mesh that describes the surface at a certain level of detail. In [Pup96] it has been shown that *all* possible meshes formed by triangles that appear in the MT are in one-to-one correspondence with consistent subsets of updates.

We denote with $\hat{S}$ the mesh corresponding to a consistent set S. Adding a node u to a consistent subset S locally refines $\hat{S}$ by applying update u^+. Deleting a node u from S locally coarsens $\hat{S}$ by by applying update u^-.

It is easy to visualize a consistent set S with a cut that divides the DAG into two parts, one of which contains exactly nodes belonging to S (see Figure 4). This cut contains those arcs of the form (u_1, u_2), where $u_1 \in S$ and $u_2 \notin S$. In the reminder, we call such a cut the *front* of set S. Thus, adding nodes to a consistent set S can be visualized as sweeping its front forward, while deleting nodes from S can be visualized as sweeping it backward.

3.4 Selective Refinement

The *front sweep* principle mentioned above is used to implement *selective refinement*, i.e., an operation that adapts the level of detail of a mesh to the needs of an application, possibly varying

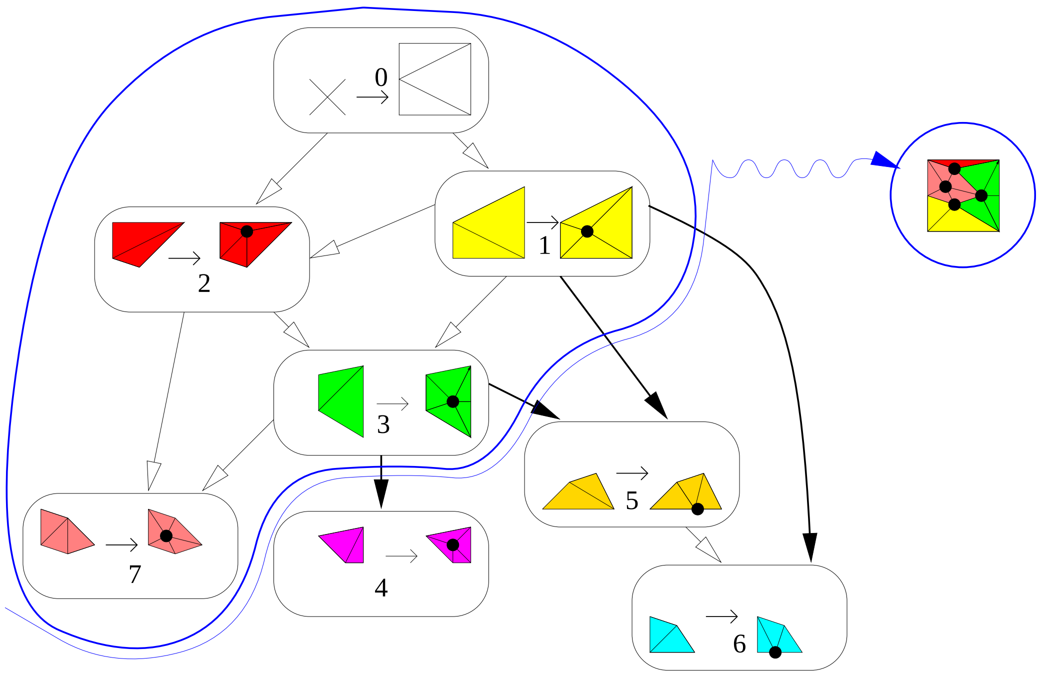

Figure 4: The blue line surrounds a consistent subset of nodes on the MT of Figure 3. The double blue line shows the corresponding front. The corresponding triangle mesh is shown in the blue circle on the right (each triangle is colored according to the update that has created it).

resolution over different portions of the mesh. Selective refinement is performed upon requirements expressed in terms of *resolution* and *size* of the output mesh.

A *resolution threshold* is a function τ assigning a real value to each triangle or group of triangles. Such a threshold is application dependent as it may depend, e.g., on the size of triangles, on their approximation error, etc. Intuitively, $\tau(t)$ measures the "signed difference" between the resolution of t and the resolution needed by the application.

Given a mesh T, we define the value of τ over T, as $\tau(T) \equiv \max\{\tau(t) \mid t \in T\}$. A value of $\tau(T) > 0$ indicates that the resolution of T is not sufficient; a value $\tau(T) < 0$ means that the resolution of T is higher than necessary. A value less than, or equal, to zero identifies triangles that satisfy resolution requirements.

A *size threshold* is simply a positive integer b which gives an upper bound to the number of triangles in the output mesh. In an application, b corresponds to the maximum number of triangles that can be processed (e.g., rendered, analyzed) with the available resources and/or within given time constraints.

Based on the resolution and size thresholds, different variants of selective refinement queries can be defined. Here, we want to find the mesh which *best approximates* a given resolution threshold τ, and has a size less or equal to a given size threshold b:

> Given an MT $\mathcal{M}$, a resolution threshold τ, and a size threshold b, find a mesh T, among all meshes associated with consistent sets of $\mathcal{M}$, such that $\#T \leq b$, and $\max\{0, \tau(T)\}$ is minimum.

This formulation gives higher priority to constraint $\#T \leq b$, over satisfaction of threshold τ. In a client-server architecture, the constraint on the maximum size of the mesh is especially relevant to allow the client to handle meshes of a size compatible with its available resources.

This scenario can be better clarified with a classical example: assume we have modeled a terrain with an MT, and we want to visualize it in the context of a flight simulator; assume that the graphical engine cannot support meshes larger than a certain size b in order not to slow down rendering. On the other hand, we would like to achieve the best visual quality within such a constraint. Then our goal is to minimize the approximation error of triangles projected onto the screen, hence we should define τ as a function that varies linearly with the projected error of a triangle. When such an error goes below a few pixels, we might decide that it is worthless to have a more accurate triangle, and set $\tau < 0$.

4 A Dynamic Algorithm for Selective Refinement

In [DFMP97, DFMP98] we have proposed and analyzed selective refinement algorithms for a general MT. Such algorithms solve a selective refinement query with a different formulation, which does not take into account the constraint on the size of the output mesh, and optimizes the number of triangles in the output.

In this section, we present a new dynamic algorithm which represents an effective heuristics for answering the selective refinement query defined in Section 3 on a Vertex-based Multi-Triangulation. The new algorithm exploits the special structure of such an MT, thus offering several advantages over our previous proposals.

The algorithm traverses the DAG while maintaining a current consistent subset S of nodes, and its corresponding mesh $\hat{S}$. Set S is progressively modified until $\hat{S}$ becomes the solution to the query.

The algorithm is *dynamic*: it starts from an existing set S, and progressively transforms it into the set S', which defines the solution of the current query. The dynamic approach is suitable for repeated queries, especially when the resolution threshold changes gradually. At startup, the current set is initialized with the root of the DAG, while the result of a given query is taken as input by the successive query. This mechanism is especially attractive in a client-server context, because initialization requires only a small mesh to be transmitted, while updates between successive queries only require a small amount of information to be sent from the server to the client.

The algorithm modifies a set S (and its corresponding mesh $\hat{S}$, accordingly) by sweeping its front forward and backward through the DAG. Sweep operations consist of adding and deleting nodes to/from S:

- Sweeping the front forward across a node u has the effect of adding u to S, and of increasing both the resolution of $\hat{S}$ and its size by two triangles; such operation is done whenever needed to decrease the value of $\tau(\hat{S})$, if allowed by the size bound b;
- Sweeping the front backward across a node u has the effect of deleting u from S, and of decreasing both the resolution of $\hat{S}$ and its size by two triangles; such operation is done whenever $\#\hat{S} > b$, or whenever it preserves $\tau(\hat{S}) \leq 0$.

The algorithm performs a front sweep in two phases:

- In the *contraction* phase, the front is moved backward: nodes are deleted from S as long as $\#\hat{S} > b$, or node deletion maintains $\tau(\hat{S}) \leq 0$;
- In the *expansion* phase, the front is moved forward: nodes are added to S as long as $\tau(\hat{S}) > 0$ and $\#\hat{S} \leq b$.

During contraction and expansion, nodes are deleted and added one at a time, according to a greedy criterion that tries to add first nodes that are likely to decrease $\tau(\hat{S})$ faster, and to delete first the nodes that are likely to increase $\tau(\hat{S})$ slower.

```
Procedure SelectiveRefinement
loop
    (τ, b) :=getQueryParams();
    contraction();
    expansion();
end loop
```

Figure 5: Pseudocode of the selective refinement algorithm. Primitive getQueryParams is used to set extraction parameters τ and b. We assume that τ, b, the current consistent set S and the current mesh are global variables.

Figure 5 shows the general structure of the selective refinement algorithm, while the contraction and expansion phases are detailed in the following subsections.

4.1 Contraction Phase

Contraction consists of iterative deletions of nodes from S, and corresponding modifications of $\hat{S}$. A node u can be deleted from S only if no child of u is in S. In this case, we say that u is a *leaf* of S. Leaves have the property that set $S \setminus \{u\}$ is still consistent with the partial order. If u is a leaf, then all triangles of u^+ are in $\hat{S}$, and deleting u from S means replacing u^+ with u^- in $\hat{S}$.

The algorithm maintains all leaves of S in a priority queue, sorted according to the increasing value of $\tau(u^-)$. The contraction phase consists of a loop in which, at each step, the node u minimizing $\tau(u^-)$ is deleted from S.

Deleting a node u may cause other nodes, parents of u, to become leaves. Such new leaves are inserted into the priority queue. The root of the MT is never inserted into the queue since a consistent set must always contain it.

The contraction loop ends when the size of $\hat{S}$ becomes less or equal to b, and no more leaves can be deleted without violating the resolution threshold τ.

The pseudocode of the contraction phase is shown in Figure 6. Variable Q is a priority queue of nodes. Primitives empty, isEmpty, insert, min, deleteMin and priority denote the standard operations on priority queues. Primitives children, parents, and root act on the DAG with the obvious semantics. Primitive delete deletes a node from set S. Primitive leaves denotes the set of leaves of S, and isLeaf is the membership test for it. Primitive coarsen performs the update represented by u^- on the current mesh.

4.2 Expansion Phase

Expansion consists of iterative insertions of nodes into S, and corresponding modifications to $\hat{S}$. A node u can be added to S only if all its parents are in S. In this case, set $S \cup \{u\}$ is still consistent. Also, all triangles of u^- are in $\hat{S}$, and inserting u into S means replacing u^- with u^+.

However, considering the nodes having all their parents in S as the only candidates for inclusion into S is not sufficient. Since any portion of $\hat{S}$ might need further refinement, any node u having at least one parent in S is a *candidate* node for expansion. When such a node u is added to S, we recursively add all its parents which are not in S, before adding u, in order to keep set S consistent with the partial order.

The algorithm maintains all candidate nodes in a priority queue, sorted on the decreasing value of $\tau(u^-)$ (in fact, value $-\tau(u^-)$ is used as priority for u in the queue). Nodes u such that $\tau(u^-) \leq 0$

Procedure CONTRACTION

```
EMPTY(Q);
for all u ∈ LEAVES(S) do
  INSERT(Q, u, τ(u⁻));
end for
while (not ISEMPTY(Q)) and (#Ŝ > b or PRIORITY(MIN(Q)) ≤ 0) do
  u := MIN(Q);
  DELETEMIN(Q);
  DELETE(S, u);
  COARSEN(u);
  for all u' ∈ PARENTS(u) do
    if (ISLEAF(S, u')) and (u' ≠ ROOT()) then
      INSERT(Q, u', τ(u'⁻));
    end if
  end for
end while
```

Figure 6: Pseudocode of the contraction phase.

are not inserted into the queue. In this case, adding u to S is not necessary to satisfy τ, while it unnecessarily increases the size of the mesh.

Inserting a node u may cause other nodes to become candidates and to be inserted into the priority queue. The expansion loop terminates when either the size of $\hat{S}$ becomes too large, or the resolution threshold is satisfied everywhere. Since each node increases the size of $\hat{S}$ by two triangles, we can always add one node, as long as $\#\hat{S} < b - 1$. Note, however, that adding a node might require adding some of its parents recursively as well. If the size threshold is violated during such operation, the algorithm stops adding nodes and exits the loop.

The pseudocode of the expansion phase is shown in Figure 7. The same primitives as in the pseudocode of Figure 6 are used. The sign of τ is changed before passing it to INSERT in order to get the element with the highest value of τ at the top of the queue. Primitive ISIN is the membership test for set S, and primitive ADD adds a node to S. Primitive REFINE performs the update represented by u^+ on the current mesh.

4.3 Standalone Implementation

A standalone version of this algorithm is implemented according to the scheme depicted in Figure 8. In this case an application loads an MT and then interacts with the user, which provides extraction criteria, to perform selective refinement. Such an implementation requires three data structures:

- A data structure for the MT, which supports primitives CHILDREN, PARENTS, and ROOT.
- A data structure for the *current mesh*, which supports operations COARSEN and REFINE.
- An *extraction status*, which handles the priority queue and supports all primitives working on current set S, namely ADD, DELETE, ISIN, LEAVES and ISLEAF.

Data structures are discussed in detail in Section 6. Here, we just discuss key issues on the implementation of primitives REFINE and COARSEN. Such primitives must use information from a node u of the MT to edit the current mesh. We exploit the special structure of vertex insertion

Procedure EXPANSION

```
EMPTY(Q);
for all u ∉ S such that PARENTS(u) ∩ S ≠ ∅ do
  if (τ(u)⁻ > 0) then
    INSERT(Q, u, −τ(u⁻));
  end if
end for
while (not ISEMPTY(Q)) and (#Ŝ < b − 1) and PRIORITY(MIN(Q)) < 0 ) do
  u := MIN(Q);
  DELETEMIN(Q);
  if (not ISIN(S, u)) then
    RECURSIVEADD(u);
  end if
end while
```

Procedure RECURSIVEADD(u)

```
for all u′ ∈ PARENTS(u) do
  if u′ ∉ S then
    RECURSIVEADD(u′);
  end if
end for
if (#Ŝ < b − 1) then
  ADD(S, u);
  REFINE(u);
  for all u′ ∈ CHILDREN(u) do
    if (not ISIN(S, u′)) and (τ(u′⁻) > 0) then
      INSERT(Q, u′, −τ(u′⁻));
    end if
  end for
end if
```

Figure 7: Pseudocode of the expansion phase and of the auxiliary procedure RECURSIVEADD, used inside it.

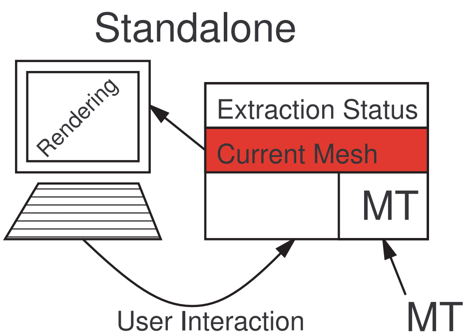

Figure 8: Selective refinement in a standalone architecture: The MT is loaded into an MT data structure, then user interaction is considered. Standalone application executes selective refinement from the MT driven by an extraction status and updates a current mesh.

and vertex removal operations to maintain the structure of a node u implicitly in the MT, while the current mesh is maintained in a topological data structure.

Refinement is possible only if the current mesh contains all triangles of u^-, and coarsening is possible only if the current mesh contains all triangles of u^+. These conditions are always guaranteed when such primitives are applied in the context of the algorithm.

Operation REFINE must replace u^- with u^+ in the current mesh. This is performed through the following steps:

1. A patch on the current mesh is found, which corresponds to u^-;
2. Such a patch is removed from the current mesh, thus leaving a hole;
3. The hole is filled with a star of triangles having v_u as central vertex: such triangles form u^+ by definition.

Step 1, called *patch identification*, is the crucial one. It is performed by accessing one edge on the boundary of the patch in the current mesh, and traversing the patch through adjacencies. A compact code stored in node u of the MT contains information to detect such an edge and to traverse the patch. The geometry of vertex v_u is also stored in u.

Operation COARSEN must replace u^+ with u^- in the current mesh. This is performed through the following steps:

1. Vertex v_u is identified on the current mesh;
2. v_u is eliminated, together with all triangles in its star (i.e., triangles of u^+), thus leaving a hole with a polygonal boundary in the current mesh;
3. The hole is triangulated with triangles of u^-.

In this case, the crucial step is the last one, which we call *boundary retriangulation*. The triangles of u^- are reconstructed by filling the hole one triangle at a time, scanning its boundary from the same edge used in patch identification, thus iteratively reducing the hole.

In Section 6.1, we give details about the coding mechanism and show that the same code used for patch identification is applied to boundary retriangulation. This mechanism is a central issue in our approach, since it highly reduces information that need to be stored in each node of the MT (and that will be transmitted in the client-server version of the algorithm discussed next).

5 Selective Refinement in a Client-Server Architecture

In a client-server version of selective refinement, two concurrent processes are running on different machines and communicating through a bi-directional transmission channel.

The server executes the same extraction procedure described in Section 4 but it does not store the current mesh, which, in this case, is maintained by the client. Each time an update must be performed, the server only modifies its extraction status, while operations REFINE and COARSEN consist of sending instructions to the client on how to update the current mesh. Some auxiliary data structures are introduced, which are necessary to relate data on the client and on the server and to improve performance by reducing communication. The client-server architecture is depicted in Figure 9, and detailed in the following subsections.

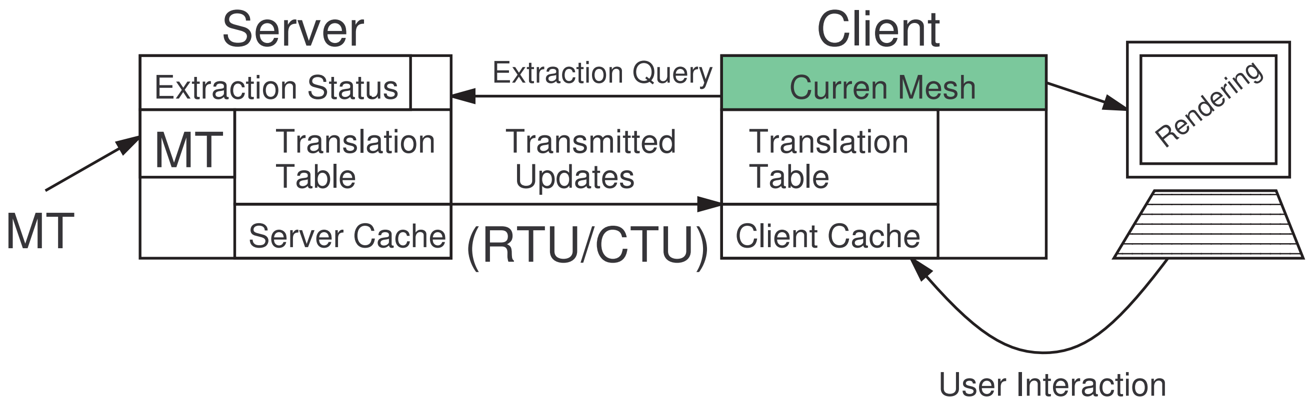

Figure 9: The client/server architecture: The MT is loaded into the server, then user interaction starts at the client. Based on it, the client sends extraction queries to the server. The server executes the extraction algorithm on the MT, making use of the extraction status and answers with a stream of Transmitted Updates (TU). The client applies the updates to the current mesh which is progressively delivered to the user. TUs are encoded/decoded through Translation Tables. A server cache and a client cache are used to implement client caching to avoid multiple transmissions of the same TU.

5.1 Roles in the Client-Server Architecture

We assume that the server process runs on a machine with sufficient memory to store an arbitrarily large MT together with all auxiliary data structures needed by the selective refinement algorithm and by the communication protocol. However, storage requirements of auxiliary structures are usually negligible with respect to those of the MT.

On the other hand, the client process runs on a machine with limited memory and computing resources. It stores a *current mesh*, which cannot exceed a given size and is used (e.g., rendered, analyzed) by a local application. The client expresses user requests in the form of an *extraction query* and sends such query to the server.

The server answers with a stream of *Transmitted Updates* (TUs). Each TU specifies a basic update operation (i.e., either a refine or a coarsen operation) on the current mesh, which will be executed by the client.

Client and server are connected through a bidirectional TCP/IP socket. This socket will carry *extraction queries* from the client and *TUs* from the server. We assume that the time spent by the server in performing selective refinement as well as the time spent by the client in updating and using the mesh is relatively negligible with respect to the time spent for communication. Under this assumption, which is realistic for most interactive applications, the socket is the true bottleneck of the architecture. Therefore, we will concentrate our efforts in reducing communication by compressing and caching TUs.

5.2 The Transmitted Updates

Extractions performed on the server are described by the pseudo-codes of Figures 5, 6 and 7, but the COARSEN and REFINE operations performed during contraction and expansion phases will have a different meaning.

Procedure GETQUERYPARAMS puts the process in a stand-by status until some extraction query is received from the client. In executing procedures CONTRACTION and EXPANSION, the server manipulates the MT and the extraction status as in the standalone version, while it generates a *Transmitted Update* (TU) each time either a REFINE or a COARSEN operation is executed.

A TU must contain all information needed by the client to modify the current mesh by *vertex insertion* or *removal*. Therefore, we have two kinds of Transmitted Updates:

- A *Refining TU* (RTU), which is generated and transmitted by primitive REFINE, instructs the client to insert a point as a new vertex in the current mesh; it contains the point coordinates, plus all information necessary to perform *patch identification* (see Section 4.3).
- A *Coarseninig TU* (CTU), which is generated and transmitted by primitive COARSEN, instructs the client to remove a vertex; it contains a reference to the vertex plus all information necessary to perform *boundary retriangulation* (see Section 4.3).

As we already mentioned, both *patch identification* and *boundary retriangulation* are represented by a unique compact code, permanently stored in each node of the MT, and filtered through an encoding/decoding process during transmission (see Section 6 for details).

Transmitted Updates are handled by two encoding primitives running on the server, and two decoding primitives running on the client:

- RTUENCODE(u) takes an MT node u and produces the RTU corresponding to update u^+;
- CTUENCODE(u) takes an MT node u and produces the CTU corresponding to update u^-;
- RTUDECODE(rtu) takes an RTU and produces the corresponding update to be performed on the current mesh;
- CTUDECODE(ctu) takes a CTU and produces the corresponding update to be performed on the current mesh.

The encoding/decoding mechanism is necessary to relate client and server data structures while maintaining the TUs as compact as possible. This mechanism will be detailed in Section 6. For now, we just assume that the results of RTUDECODE(RTUENCODE(u)) and CTUDECODE(CTUENCODE(u)) provide information sufficient to perform updates u^+ and u^- on the current mesh at the client, respectively.

5.3 Caching

During client-server interaction, it may happen that certain details of the mesh are first refined, then coarsened, and then refined again. In a straightforward implementation, this implies that all details must be (re)transmitted for every update. In order to reduce channel traffic, a mechanism is necessary such that the client can "remember" updates it has received in the past.

An attractive idea is that of building a local copy of the MT on the client, using just the updates that have been transmitted so far. Then, the selective refinement algorithm running on the server is mirrored on the client based on the local MT. When a portion of the DAG must be visited, which is not present on the client, the server sends the corresponding information to the client. On the other side, the client process stops and waits for information from the server in order to proceed. This solution however has some drawbacks. First of all, the amount of memory needed to store the local MT on the server is not bounded a-priori, against our assumption of a client with a limited memory on board. Moreover, mirroring the selective refinement algorithm presents some intrinsic difficulties because nodes inserted in the priority queue during refinement may neither be part of the local MT, nor be consistently added to it.

In this paper, we propose a simpler solution to reduce the channel traffic based on caching, which already gives good results. The client maintains some of the TUs it receives in a local cache, called the *client cache*, with a limited capacity (say, K updates). We do not define the policy adopted for removing updates when the cache is full and a new update needs to be stored. We just assume that cache capacity and its management policy are known to the server.

Each entry in the client cache contains sufficient information to support both a refining update and its corresponding coarsening update for a given node u. In fact, we will see in Section 6 that such information can be generated locally from a new TU received from the server, independently on whether it is an RTU, or a CTU.

The server maintains a reference copy of the client cache, called the *server cache*, managed with the same policy. The server cache contains references to MT nodes rather than information on the corresponding updates. Looking into its cache, the server knows the contents of the client cache at each instant of time. The server queries its cache when it traverses a node u to decide whether a complete TU or a just reference to it in the client cache must be sent.

The server cache implements the following primitives:

- MEMBER(u): test whether an MT node u is in cache;
- ADDNODE(u): add a new node to the cache (also managing cache overflow);
- GETCODE(u): get the cache key ck for a given MT node u.

The client cache implements the following primitives:

- ADDUPDATE(upd): insert the description upd of an update in the cache (also managing cache overflow);
- GETUPD(ck): used by the client to get the update corresponding to a given cache key ck.

The client and server caches are managed coherently by the client and server processes: for each node u cached by the server, we are always guaranteed that GETUPD(GETCODE(u)) will return update information regarding node u at the client.

An MT node u can be transmitted as a complete RTU, by sending RTUENCODE(u), as a complete CTU, by sending CTUENCODE(u), or as a cached RTU/CTU, by sending its cache key GETCODE(u).

This transmission scheme requires one bit to enable the client to distinguish whether the TU is cached or not. Refining and coarsening TUs are distinguished based on their position in the stream: first all CTUs (cashed or not) are sent, then a stop code, followed by all RTUs (cached or not), and a final stop code. The stop code is just a placeholder to cope with lookahead initialization in the client process. It can be coded, for instance, as an RTU / CTU with a null order. The cache-aware version of primitives REFINE and COARSEN are detailed in Figure 10.

5.4 The Client Process

The algorithm running on the client is very simple. It sends an extraction query (in terms of τ and b as defined in Section 3.4) to the server, then waits for a sequence of CTUs, terminated with a stop code, followed by a sequence of RTUs, terminated with a stop code. Each CTU/RTU is decoded and applied to the current mesh. The pseudocode is shown in Figure 11. Note that transmission operations **transmit** and **receive** can be either synchronous or asynchronous. We assume only that messages are delivered in the order they are sent and that no message is lost.

While primitives REFINE and COARSEN on the server are in charge of just transmitting updates, primitives REFINE and COARSEN on the client operate directly on the current mesh, in a similar way

```
Procedure REFINE(u)
if MEMBER(u) then
    transmit CACHED_TU(GETCODE(u));
else
    ADDNODE(u);
    transmit RTUENCODE(u);
end if
Procedure COARSEN(u)
if MEMBER(u) then
    transmit CACHED_TU(SC.GETCODE(u));
else
    ADDNODE(u);
    transmit CTUENCODE(u);
end if
```

Figure 10: Primitives REFINE and COARSEN for the client-server architecture with cache.

as described for the standalone version (see Subsection 4.3). Here, they take as input an update obtained from decoding, rather than its corresponding MT node. The client process is detailed in Figure 11. By reading codes of the client process together with those of REFINE and COARSEN procedures that run on the server, it is easy to see that nodes and their corresponding updates are added to the two copies of the cache whenever a (non-cached) transmission occurs. Therefore, ADD operations will be performed in the same order at the server and at the client, hence maintaining the server and the client caches coherent.

6 Implementation Issues and Data Structures

The algorithm for selective refinement requires the following data structures (see Figure 9):

- The *MT* is a data structure maintained by the server to manage the Multi-Triangulation. Update codes are maintained at the nodes of the MT, which are used during traversal to generate Transmitted Updates.
- The *extraction status* is also maintained by the server.
- The *current mesh* is maintained by the client and is updated through REFINE and COARSEN operations.
- The *translation tables* are two symmetric data structures, one maintained by the server, and the other by the client, which are used during encoding of TUs to relate data structures on the two processes.
- The *server cache* and the *client cache*, maintained by the server and by the client, respectively, are used to filter TUs before encoding and decoding.

A central issue that relates different data structures is the internal representation scheme that we have developed for updates. Therefore, we first discuss this issue, then we will describe each of the above data structures in detail.

Client Process

```
Initialize;
loop
  sendQueryParams(τ, b);
  receive msg;
  while msg ≠ STOP_CODE do
    if ctu = CACHED_TU(ck) then
      upd :=getUpd(ck);
    else
      upd :=CTUdecode(ctu);
      addUpdate(upd);
    end if
    coarsen(upd);
    receive msg;
  end while
  receive msg;
  while msg ≠ STOP_CODE do
    if rtu = CACHED_TU(ck) then
      upd :=getUpd(ck);
    else
      upd :=RTUdecode(rtu);
      addUpdate(upd);
    end if
    refine(upd);
    receive msg;
  end while
end loop
```

Figure 11: Pseudocode of the selective refinement algorithm running on the client. Primitive sendQueryParams sends the extraction parameters τ and b to the server. Primitives CTUdecode and RTUdecode decode a coarsening and a refining TU, respectively, and return an update. Primitives refine and coarsen apply an update to the currrent mesh.

6.1 Representation of Updates

In Section 3, we have defined an update corresponding to a node u as a pair of *patches* u^- and u^+. Instead of listing triangles of such patches explicitly, we propose a compact representation, which is based on the possibility of encoding a triangulated simple polygon through a bit stream, plus some geometrical information that can be retrieved from the current mesh.

The encoding/decoding method we propose is valid in general as a representation of a triangulated simple polygon. The interested reader may refer to [DFMMP99b] for details and a formal proof of correctness. Here, we just describe its application to represent updates on the MT, and to perform such updates on the current mesh. For the sake of brevity, we describe only the general case of vertex v_u being an internal vertex on the current mesh. The special case of vertex v_u being on the boundary of the current mesh can be also treated with the same mechanisms described below without increasing storage and computational load (see [DFMMP99a] for details).

As remarked in Section 4.3, crucial steps during update operations on the current mesh are: how to identify a patch in an existing mesh (*patch identification*); and how to put a patch into a polygonal

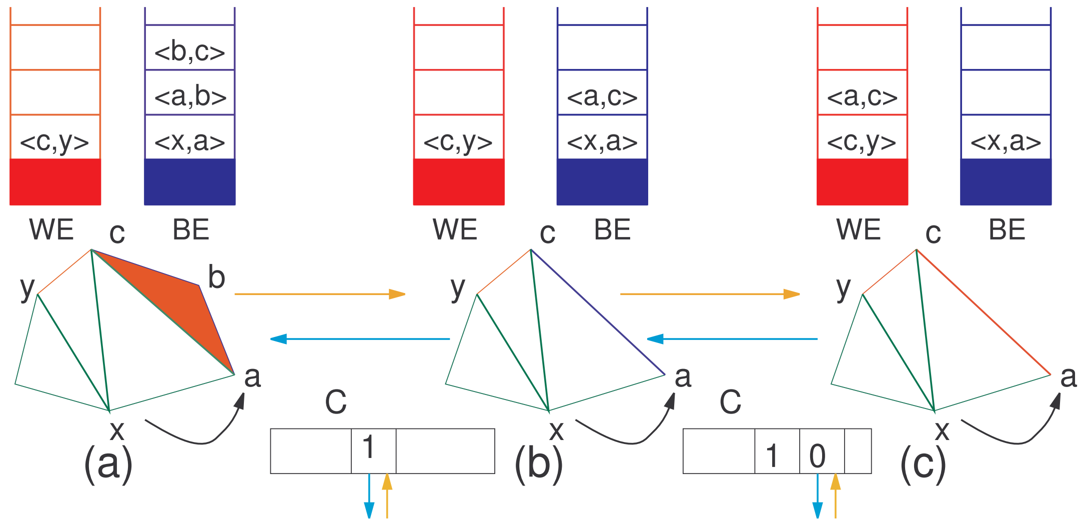

Figure 12: The three phases in ear cutting encoding. Orange arrows depict flow during encoding. Light blue arrows show flow during decoding. During encoding, from (a) to (b) red triangle is cut, point b is added to G and 1 is added to C. Blue edges $<a,b>$ and $<b,c>$ pop out of the stack WE and $<a,c>$ is pushed into it. From (b) to (c) $<a,c>$ pops out of WE into BE and 0 is written into C. The meaning of edge colors is as follows: red edges are in BE, blue edges are in WE.

hole (*boundary retriangulation*). In both cases, the patch is a triangulated polygon (corresponding to an update u^-) which must be either removed or used to fill a hole.

6.1.1 Encoding a node

It is easily understood that both updates corresponding to a node u are fully specified by maintaining the structure of u^-, plus the new vertex v_u inserted by u^+.

In order to encode u^-, we traverse it by simulating an *ear-cutting* procedure, which progressively cuts off triangles and reduces its boundary polygon, until no more triangle is left. While performing such a procedure, we generate a bit stream C, which traces all steps of the ear-cutting.

The algorithm uses two stacks of edges called BE (Boundary Edges) and WE (Working Edges). BE can contain only edges that belong to the boundary of u^-, and WE can contain edges of u^- that are analyzed in pairs in order to find ears to cut.

The encoding algorithm works as follows:

- **Initialization:** BE and WE are initially empty; push all edges of the boundary of u into BE so that the boundary with a consistent orientation can be obtained by reading the stack from the top.

- **Step:** If on the top of WE we have $<b,c>$ followed by $<a,b>$, and if u^- contains the triangle $<a,b,c>$, then we can cut an ear: delete the two edges from WE; push $<a,c>$ on WE; output 1 to C (see Figure 12 from (a) to (b)).

  Otherwise output 0 to C, pop the first edge from BE and push it on WE (see Figure 12 sequence (b) to (c)).

- **Termination:** If BE is empty and WE holds on top $<b,a>$ and then just $<a,b>$ then stop.

Moving edges from BE to WE is equivalent to walk around the current polygon, while popping pairs of edges from WE corresponds to performing an ear cut: whenever an ear is cut, two edges are removed from the current polygon, and a new edge is inserted into it at their place. An ear cutting operation causes 1 to be added to C. When we cannot perform ear cutting we walk around the current polygon, and the procedure outputs 0 to C.

We save the bitstream C, plus the edge $<a,b>$ found in the stack at termination of the procedure (hereafter called the *starting edge*). Edge $<a,b>$ is coded by a pair $<v,k>$, where: v is a reference to the vertex, either a or b, whose insertion in the MT created $<a,b>$; and k is an order number of $<a,b>$ as seen around v when it is inserted into the MT. Note that this is just a way to give a unique name to all edges of the MT, which is cheaper than maintaining explicit references to vertices a and b. Since we always build the MT by guaranteeing that the degree of each vertex is bounded by a small constant, index k requires few bits.

In summary, update information at node u is given by a tuple $<d, C, <v,k>, v_u>$, where:

- d is the number of vertices on the boundary of u^- (which corresponds to the degree of the new vertex v_u in u^+);
- C is the bit stream generated by the encoding algorithm;
- $<v,k>$ is the starting edge, encoded as described above;
- v_u is a reference, in the MT data structure, to the vertex to be inserted by u^+.

6.1.2 Use of codes in refinement

Patch identification is performed by traversing an existing patch u^- on the current mesh, through a decoding algorithm which makes use of code C, starting edge $<a,b>$, and the two stacks BE and WE used during encoding. The decoding algorithm corresponds to running the steps of the encoding algorithm in reverse order. In fact, code C is read from its last to its first element.

The decoding algorithm works as follows:

- **Initialization:** BE and WE are initially empty. Push the starting edge $<a,b>$ stored at node u (obtained from its code $<v,k>$) and its opposite $<b,a>$ into WE.
- **Step:** Read a bit from C.

  If the bit is 0, then the top of WE is a boundary edge of u^-: pop an edge from WE and push it into BE (see Figure 12 sequence (c) to (b)).

  If the bit is 1, then pop an edge $<a,c>$ from WE; look in the current mesh for the triangle $<a,b,c>$ bounded by $<a,c>$ and mark it as part of u^-; push $<a,b>$ and then $<b,c>$ into WE. (see Figure 12 sequence (b) to (a)).
- **Termination:** WE is empty; C has been read completely. At this point, reading BE from the top we find the oriented boundary of M.

After this operation we obtain a list of marked triangles to be deleted form the current mesh, as well as the boundary of u^- in the stack BE. It is easy to create the new patch u^+ by connecting vertex v_u to all vertices of the boundary.

6.1.3 Use of codes in coarsening.

We perform boundary retriangulation by simulating the encoding algorithm.

Vertex v_u is found in the current mesh, and it is deleted from it together with its incident triangles and edges. This gives us also the boundary of u as a sequence of edges.

We use this boundary to load stack BE the way it was at the beginning of the offline encoding process that produced bitstream C. In order to do this, we just have to push edges of the boundary beginning at the starting edge $<a, b>$ (obtained from $<v, k>$ as above), followed by the other edges of the boundary in the conventional order.

Then we assume that the encoding algorithm must produce C. Bits are read from C, rather than written into it; and ears are added to fill the current polygon, rather than cut from it. At each step, we read a bit from stream C. This is obtained by modifying the step of the encoding algorithm as follows:

- **Step:** Get a bit from C.

  If it is a 1, then on the top of WE we have two edges $<b, c>$ and $<a, b>$ forming a triangle $<a, b, c>$, which is added to the current mesh.

  Otherwise, pop the first edge from BE and push it into WE.

The termination step of the encoding algorithm in this case does not produce any output.

6.2 Encoding the Multi-Triangulation

The Multi-Triangulation is encoded through a data structure implementing the DAG, together with its traversal primitives: CHILDREN, PARENTS and ROOT. The structure consists of three arrays, representing vertices used in the MT, and nodes and arcs of the DAG, respectively:

- The entry for a vertex v stores its coordinates plus possible attributes.
- The entry for an arc a stores the indexes of the source and destination nodes of a, and the index of the next arc having the same destination node.
- The entry for a node u stores the index of the first arc outgoing from u; the number of arcs outgoing from u; the index of the first arc entering u. *Update information* is also stored, in the form described in Subsection 6.1. In the remainder of the paper, we call such information a *node update*. Note that vertex v_u corresponding to the node will be stored as an index to the vertex array. Moreover, we might store additional information, depending on applications (e.g., the approximation error of the triangles of u^- and u^+), which might be used to evaluate threshold τ.

Special care must be taken for the root node, which has a more complex structure. We encode the root with a compressed data structure that we proposed in [DFMP99], which is not detailed further here. However, the root is always a small mesh, therefore we will disregard its cost.

6.3 The Extraction Status

The *extraction status* contains all information necessary to traverse the MT: the current consistent set S of nodes and a priority queue of nodes. The latter is a standard data structure and will not be detailed further.

The extraction status represents in the server the current consistent subset S of MT nodes, and supports primitives isIn, add, delete, isLeaf, and leaves.

We maintain leaves in a dynamic structure supporting linear scanning and dynamical insertion/deletion of elements. Linear scanning is applied to implement primitive leaves.

Moreover, we keep a counter for each node u of the MT, which is used to determine the situation of u with respect to set S: if $u \notin S$, then the counter is equal to zero; if $u \in S$, then the counter is equal to one plus the number of children of u which are in S.

With this convention, function isIn reduces to testing whether the counter of a node u is equal to, or different from zero; function isLeaf reduces to testing whether such counter is equal to one. Primitives add and delete update both the node counters and the list of leaves.

Note that the number of children of a node is bounded by the degree of its central vertex, which is bounded by a small constant. Thus, the counter associated with each node fits in few bits. The size of the list of leaves is bounded by the size of the current mesh, since each leaf u contributes to it with all triangles of u^+. On average, the number of leaves is expected to be about $\frac{\#\hat{S}}{\delta}$, where δ is the average degree of a vertex (in practice, $\delta = 6$).

6.4 Node Updates and Transmitted Updates

The client must update its local mesh by performing refine and coarsen operations based on *update information* communicated by the server.

Note that we cannot transmit a node update to the client as it is stored in the MT structure of the server (see Section 6.1): references to vertices v and v_u are relative to the MT data structure, while we must address copies of such vertices in the client. More precisely, v references a vertex in the current mesh, while v_u (which is used only in the refinement update) must be created as a new vertex in the current mesh.

The server will transform node updates into *Transmitted Updates* (TUs) prior to transmission, through an encoding process, in order to make them usable by the client. The coding scheme for TU is actually different whether we want to obtain a Refinement TU (RTU) or a Coarseninig TU (CTU).

In both cases, however, the key issue is to obtain a common numbering for vertices on the two processes.

6.4.1 Translation Tables

Translation tables implement vertex numbering (i.e. a bijection between a set of vertices and a set of indices), which is common between the server and the client (see Figure 9). Tables are managed in a coherent way, meaning that both tables are exposed to the same sequence of insertions and deletions of vertices. In this way, we always guarantee that the two copies of the same vertex, at the server and at the client, will receive the same index.

This relation is implemented via hashing: the same hashing function and a common key is used by both the client and the server. We use the vertex coordinates of a vertex as a key, since they must be necessarily transmitted to the client the first time a vertex is inserted in the current mesh.

The translation table have a garbage collection mechanism, such that the keys of vertices which are no longer referenced (either in the current mesh, or by nodes in cache) are automatically released. Details on the implementation of translation table are contained in [DFMMP99a].

The two hash tables have the same size W, which must be chosen according to a convenient *load factor* α (e.g., $\alpha \sim 0.1$). A vertex may be maintained in the table because it is referenced either by the current mesh, or by some update in the client cache (see Sections 5.3 and 6.5). If b is the size threshold, then the maximum number of vertices in the current mesh will be $n \simeq b/2$. If we denote with K the size of the cache, then the number of vertices currently referenced is bounded from above by $\max(n, 2K)$. Therefore we can take $W \sim \alpha \max(n, 2K)$.

6.4.2 Coding and Decoding Transmitted Updates

Translation tables are used by the encoding/decoding procedures for TUs, already introduced in Section 5: RTUENCODE, RTUDECODE, CTUENCODE and CTUDECODE.

Procedure RTUENCODE takes as input a node update $<d, C, <v, k>, v_u>$ and generates a corresponding RTU $<d, C, <i_v, k>, P_u>$ where i_v is the index of vertex v obtained from the translation table, while P_u contains the coordinates of v_u (plus, possibly, additional vertex information, such as color and vertex normal, depending on the application). Moreover, it inserts v_u in the translation table (if it is not already present in it).

Procedure RTUDECODE takes as input an RTU $<d, C, <i_v, k>, P_u>$ and generates a corresponding tuple $<d, C, <a, b>, v'_u>$ where v'_u is the client copy of v_u, obtained by allocating one new vertex and associating information P_u with it. Edge $<a, b>$ of the current mesh is obtained from $<i_v, k>$ as follows: first, the copy v' of vertex v on the current mesh is obtained from the translation table at index i_v. Then the star of edges incident at v' is scanned until an edge marked with $<i_v, k>$ is found. Vertex v'_u in inserted in the translation table (if it is not already present in it).

Procedure CTUENCODE takes as input a Node Update $<d, C, <v, k>, v_u>$ and generates a corresponding CTU $<d, C, h>$ where h is the index of vertex v_u obtained from the translation table.

Procedure CTUDECODE takes as input a CTU $<d, C, h>$ and generates a corresponding tuple $<d, C, <a, b>, v'_u>$, where v'_u is the client copy of v_u, found by h in the translation table, while $<a, b>$ is the first edge found in the boundary of the star of v'_u in the current mesh.

6.4.3 Encoding the Current Mesh

The data structure used to maintain the current mesh must support addressing its vertices and edges explicitly. For each vertex, we must be able to retrieve all its incident edges in some order (either clockwise or counterclockwise). For each edge, we must be able to retrieve its two incident triangles and we must be able to understand it is a boundary edge if one such triangle does not exist. We must also be able to edit the data structure by adding new vertices, edges, and triangles. Thus, we have chosen to implement the current mesh with the DCEL data structure [PS85] which supports all such operations efficiently.

Some additional care must be taken to support conventions on edge labeling and the coding scheme for the starting edge. In both cases, the key point is to list edges around a given vertex starting always from the same edge. We describe here how this can be obtained for edges introduced by refinement operations. This mechanism can be extended easily to edges of the root mesh. When we execute a refinement operation, we start inserting edges incident at the new vertex v_u by putting edge $<v_u, a>$ in the first place, where $<a, b>$ is the starting edge used in patch identification. This solves the problem of finding the starting edge in coarsening operations.

Proceeding from this edge to the remaining edges in the star of v_u, in conventional order, each edge is given an order number, and is uniquely identified by a pair $<v_u, k>$, which is used afterwards to encode starting edges in refinement updates. Since a refinement operation is simulated each time we produce (off-line) the representation of a node update, we are sure that labeling of edges at the client will be consistent with that stored at nodes of the MT.

Under the above assumptions we can specify the mesh operations necessary to implement REFINE(*upd*) and COARSEN(*upd*) on the current mesh, where *upd* is an update of the form $<d, C, <a, b>, v'_u>$

(see Figure 13).

REFINE(upd)
$< d, C, < a, b >, v'_u > := upd$
identify patch M using d, C and $< a, b >$;
delete triangles of M,
insert v'_u as a new vertex of the current mesh;
create the star of v'_u in the conventional order, starting at $< v'_u, a >$ and labeling edges by an increasing index;

COARSEN(upd)
$< d, C, < a, b >, v'_u > := upd$
delete v'_u and its star;
collect hole poligonal boundary Q
retriangulate Q using d, C and $< a, b >$;

Figure 13: Implementation of REFINE and COARSEN procdures at the client.

6.5 Implementing Server and Client Caches

We assume that the client maintains a cache containing some of the updates already received. The client cache has a limited capacity, say K updates. The server also maintains a copy of this cache, and a common policy (e.g., FIFO) is adopted for handling cache overflows both at the client and at the server.

For a node u, the cache stores some information that is sufficient to describe the update the current mesh must undergo when u is either refined or coarsened. The server just stores a reference to the corresponding node in the MT.

In the following, we give some implementation detail about primitives ADDNODE and ADDUPDATE that we already outlined in Section 5.3. Primitives GETUPD, MEMBER and GETCODE have a straight-forward implementation and will not be detailed further.

Primitive ADDNODE is used at the client each time a complete TU (either RTU or CTU) is transmitted. This adds a reference to the corresponding node u to the server cache (possibly discarding an overflow if the cache is full). Primitive ADDUPDATE is used symmetrically by the client to insert a decoded update in the client cache. Note that, although in the encoded format an RTU contains more information than its corresponding CTU, decoded RTUs and CTUs have exactly the same format, hence a cached update can be reused both in refining and in coarsening. An additional bit is used in each cache entry to indicate whether vertex a or vertex b of the starting edge $< a, b >$ was referenced in the encoded format.

According to the straight-forward one-to-one correspondence between updates and nodes, the server cache and the client cache are coherently managed, i.e., they are exposed to two corresponding streams of calls $\{$ADDUPDATE$(upd_i)\}$ and $\{$ADDNODE$(u_i)\}$ where update upd_i corresponds to node u_i (compare the client process in Figure 11 and the algorithm in Figure 10).

Transmitted Updates contain an initial bit to discriminate between cache references and full TUs. Cache references in cached updates will take $\log_2(K)$ bits where K is the cache size. Refining TUs are discriminated from Coarsening TUs simply by the order they come (see the client process in Figure 11).

TU type	flag	degree d	code C	vertex ref. h	index k	vertex coord	cache ref.	total
RTU	1	4	$2(d-1) \leq 20$	16	4	96	–	$119 + 2d \leq 141$
CTU	1	4	$2(d-1) \leq 20$	16	–	–	–	$19 + 2d \leq 41$
cached	1	–	–	–	–	–	$\log_2 K$	$1 + \log_2 K$

Figure 14: Number of bits for a complete RTU $=<C, d, <h, k>, P_u>$, a complete CTU $=<C, d, h>$, and for a cached TU, according to the assumption that the degree of a vertex is $d \leq 11$ and the mesh size is $\#\hat{S} \leq 4000$ triangles. K denotes the capacity of the cache.

7 Experimental Evaluation

The client-server system is still under implementation. In this Section, we report some results about the traffic generated on the channel, obtained through simulations performed on our standalone implementation.

For our experiments, we have used the following assumptions, which are summarized in Figure 14: the maximum degree d of a vertex is ≤ 11, thus it is represented on four bits; the length of code C for a TU is $\leq 2(d-1) = 20$ bits; the bound b on the size of the extracted mesh is 4000 triangles, i.e., about 2000 ($< 2^{11}$ bits) vertices, therefore the translation tables use addresses on 16 bits; vertex coordinates are floating point numbers of 32 bits each.

We use an MT representing the Devil Peak terrain dataset, made available from the U.S. Geological Survey. The Multi-Triangulation is built on raw data consisting of a regular grid of 165×231 elevation data, by using a vertex decimation algorithm described in [DFMP97]. The mesh at full resolution has 75440 triangles, the root mesh has two triangles, and the whole MT contains 38113 nodes spanning all intermediate resolutions.

We considered a terrain exploration in which the viewpoint moves along a trajectory, and the surface representation is changed dynamically. At each frame, at most 4000 of such triangles take part in the current mesh which is visualized on the client. Such a current mesh is dynamically updated based on a view-dependent threshold τ: at each frame, inside the view frustum, the resolution is maximum near the viewpoint, and decreases linearly with the distance from the viewpoint; outside the view frustum, any resolution is accepted. We experimented three trajectories for the viewpoint: a linear trajectory, a circular trajectory, and a random trajectory.

The results reported here do not take into account the creation of the first frame, which involves traversing the MT from the root and sending all the RTUs necessary for the first pair (b, τ). We consider just a navigation setting in which we have a current frame and we modify it into the next one. A snapshot of the simulation is shown in Figure 15.

Two groups of experiments have been performed. The first group is concerned with the correlation between the number of bits sent from the client to the server at each frame, and the distance between the locations of the viewpoint at two successive frames. In all such experiments, caching is disabled. Results are shown in Figure 16. As expected, a more gradual change in the viewpoint position results into fewer updates from frame to frame: as the distance between two successive locations becomes larger, the length of the bitstream increases. For the rectilinear and circular trajectories, the standard deviation is roughly increasing with such distance. For the random trajectory, the standard deviation is lower than for the other two trajectories, and it is not so sensitive to the length of the step in the viewpoint path.

The second group of experiments studies the impact of the cache size on the length of the bit-

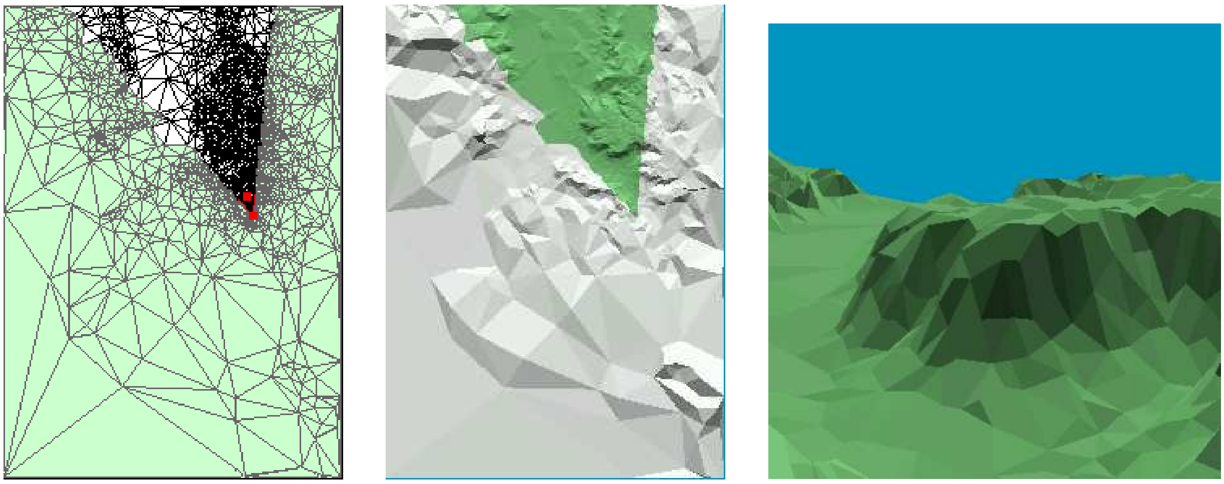

Figure 15: One of the meshes extracted during the experiments: top view (wireframe and solid, the view cone is highlighted), and eye view.

stream. Our cache is a queue managed in a FIFO way: when the cache is full, the oldest element is eliminated. A small step in the movement is used for a more gradual change in the mesh. Results are shown in Figure 17 for different sizes of the cache. In the last line of each table, the cache is large enough to contain all updates ever used during the process. In this case, every RTU is sent with its complete description exactly once; successive RTUs, and all CTUs, are found in the cache.

From the experiments, it seems that a reasonably small cache (holding about 1000 - 2000 nodes) works well. With a larger cache, more updates are transmitted just as an address, but longer addresses are needed. Thus, larger caches do not always give shorter bitstreams. For an optimal cache size, we get a traffic of about 0.5 – 1Kb per frame, which, in the assumption of 15 frames per second, gives about 7.5 – 15Kb per frame. This is a reasonable rate for web transmission.

If we are interested in evaluating the channel traffic vs the cache size, we can write the average number of bits per update as:

$$4 + \frac{\#CR}{\#RTU + \#CTU}\left(\frac{\#RTU}{\#CR}\|RTU\| + \frac{\#CTU}{\#CR}\|CTU\| + \log_2 K\right)$$

where $\|RTU\|$ is the number of bits to code a complete RTU and $\|CTU\|$ is the number of bits to code a complete CTU, and $\#CR$ denotes the number of TUs sent as cache references. Now if the cache size K increases, cache miss rates decrease and, therefore, $\frac{\#RTU}{\#CR}$ and $\frac{\#CTU}{\#CR}$ will tend to zero. In this case, the average length of an update will tend to $4 + \log_2 K$ that is a (slowly) increasing function. On the other hand, if the cache capacity tends to zero, the average update length will tend to:

$$\frac{\#RTU\|RTU\| + \#CTU\|CTU\|}{\#RTU + \#CTU}$$

So, in between there should be an optimal cache length that minimizes update average length and optimizes channel traffic. Refined policies can adaptively tune the cache size to obtain best throughput.

With respect to no caching, the length of the bit stream is about 11%, 18%, and 10% for a rectilinear, a circular, and a random trajectory, respectively. The random trajectory needs much smaller caches: a cache size of 3000 nodes is already enough to find all necessary updates in cache. Yet, a smaller cache (e.g., 1000 nodes) is more convenient since fewer bits are sufficient to address its elements and thus give shorter bitstreams.

step	updates	RTUs per frame	CTUs per frame	bits per frame	standard deviation
0.5	559.37	551.96	7.40	66943	10305
1.0	574.75	561.27	13.47	68197	10430
1.5	588.56	568.92	19.64	69252	10516
2.0	602.87	577.02	25.85	70363	10625
2.5	616.92	585.10	31.81	71466	10721
3.0	630.13	592.61	37.51	72494	11098
3.5	640.83	598.35	42.47	73292	11013
4.0	656.91	607.89	49.01	74584	11117
4.5	666.72	612.64	54.07	75265	11148
5.0	677.24	618.48	58.75	76071	10941

Rectilinear trajectory

step	updates	RTUs per frame	CTUs per frame	bits per frame	standard deviation
1.0	254.15	232.42	21.73	28579	20806
2.0	298.12	258.34	39.78	32094	21472
3.0	340.47	282.86	57.61	35436	22102
4.0	384.12	308.83	75.29	38949	22603
5.0	424.66	331.47	93.19	42065	22975
6.0	454.53	347.51	107.02	44296	23191
7.0	500.88	375.44	125.43	48063	23487
8.0	542.89	400.06	142.83	51406	23392
9.0	566.48	413.10	153.37	53206	23955
10.0	609.25	437.61	171.64	56555	24068

Circular trajectory

step	updates	RTUs per frame	CTUs per frame	bits per frame	standard deviation
0.5	376.59	371.38	5.21	45046	6992
1.0	340.95	332.09	8.86	40369	6498
1.5	345.15	331.62	13.53	40410	6267
2.0	361.98	343.19	18.78	41921	6772
2.5	382.48	356.99	25.49	43731	6982
3.0	411.48	378.26	33.21	46467	9445
3.5	437.63	398.75	38.87	49066	9023
4.0	451.79	406.88	44.90	50176	8452
4.5	469.69	423.10	46.58	52173	8469
5.0	506.29	450.66	55.63	55698	9362

Random trajectory

Figure 16: Results of simulation for a moving viewpoint with different steps of movement. Caching is disabled. All quantities shown refer to their mean value per frame. The last column is the standard deviation of the number of bits per frame.

For the circular trajectory and small caches, more CTUs are not found in cache. This happens because they are reused after a complete round trip of the viewpoint, while they have been dropped from the cache in the meanwhile.

Statistical analysis of the experimental data shows that there is a strong *linear* relation between added cache slots and cache hits. In particular, the number of RTUs found in cache linearly increases with the number of cache positions. In order to increase the number of found RTUs by one unit, we must add 1.62 cache entries. This relation is extremely stable, and, in our experiments, this brought quickly cache to saturation and to optimal operations for a cache size of 1000 positions: no more RTU hits comes from adding new cache positions.

8 Concluding remarks

The client-server algorithm for selective refinement proposed in this paper has several innovative aspects. Simulations show that it can be used effectively and efficiently on reasonably large geometric models, not only in the context of a (relatively fast) local network, but even on a (relatively slow) geographic network, like the Web.

While the standalone version is already implemented, we are now proceeding with the implementation of the complete client-server version. We foresee that further improvement on bandwidth occupancy can come from some blind, ZL-like [ZL77], compression of updates once they are grouped in batches.

Several side issues that we had to face in designing the algorithm are interesting results on their own.

The mechanism for representing updates is in fact a compressed data structure for encoding the connectivity of triangulated polygons with just one bit per edge. Such a mechanism can be further improved by noting that the bitstream C has always a tail of zeros, therefore it could be shortened by considering only its shortest prefix that contains all its 1 digits. By using a mechanism based on lookup tables (computed once off-line), it is always possible to choose the starting edge that gives the longest tail, thus minimizing the length of the prefix.

Moreover, this mechanism allowed us to design a multiresolution data structure that is largely more compact than those described in our previous work, and it is competitive with other recent proposals [Hop98, ESV99]. Under the assumptions of Section 7, the MT corresponding to a high resolution mesh with 2^{16} vertices can be encoded with just 340 bits per vertex, i.e., with just about 75% overhead with respect to the plain mesh represented with an indexed data structure.

The encoding/decoding mechanism of updates based on translation tables is a non-trivial mechanism for relating geometric data structures that are resident on different machines. By following the same approach, it is easy to modify our algorithm and encoding mechanism to deal with multiresolution models based on vertex splits, rather than vertex insertion.

Caching allowed us to obtain a surprisingly high reduction of bandwidth occupancy with relatively small additional storage requirements. It would be easy to devise a mechanism that automatically sizes the cache depending on the size threshold and on memory constraint at the client.

As mentioned in Section 5.3, a more complex form of caching is also possible, in which the client keeps a partial copy of the MT. Such copy is built while progressive transmission occurs by collecting the updates received so far. In this way, part of the work for selective refinement could be performed autonomously by the client, thus further reducing the number of transmitted updates. The communication from the server to the client would be limited to the transmission of new updates, never seen by the client before; thus, each update would be sent just once, as an RTU. In order to do this, we need a more sophisticate synchronization protocol between client and server. We plan to face this issue in the future.

The client-server algorithm can be directly extended to the case of a server and several clients. In this case, the server just has to keep one copy of additional structures (traversal status, translation table, and server cache) for each client, while the same copy of the MT can be used for serving them

cache size	RTUs in cache	RTUs out cache	CTUs in cache	CTUs out cache	bits per frame	standard deviation
0	0.00	551.96	0.00	7.40	66943	10305
100	89.92	462.04	0.00	7.40	56781	8469
200	113.12	438.84	0.00	7.40	54273	7818
300	133.16	418.80	0.00	7.40	52162	11386
400	373.78	178.18	0.56	6.83	25447	10624
500	479.88	72.08	1.11	6.28	13664	8463
600	519.83	32.13	1.46	5.93	9747	4342
800	534.58	17.38	1.83	5.56	8120	2829
1000	539.06	12.90	2.28	5.11	7624	2144
2000	543.77	8.18	4.07	3.32	7635	1580
3000	543.97	7.99	5.74	1.65	8148	1594
6000	543.98	7.98	7.40	0.00	8685	1647

Rectilinear trajectory

cache size	RTUs in cache	RTUs out cache	CTUs in cache	CTUs out cache	bits per frame	standard deviation
0	0.00	232.42	0.00	21.73	28579	20806
200	73.69	158.72	0.89	20.84	20315	18274
400	165.77	66.64	2.54	19.19	10150	12755
600	200.05	32.36	4.09	17.64	6531	5527
1000	209.12	23.29	5.44	16.28	5521	4060
2000	212.79	19.62	8.94	12.79	5303	3492
3000	212.97	19.44	16.30	5.43	5448	3480
4000	213.00	19.41	19.98	1.75	5415	3470
5000	213.14	19.27	20.85	0.88	5626	3495
6000	213.10	19.31	21.27	0.46	5627	3521
15000	215.42	16.99	21.73	0.00	5613	3509

Circular trajectory

cache size	RTUs in cache	RTUs out cache	CTUs in cache	CTUs out cache	bits per frame	standard deviation
0	0.00	371.38	0.00	5.21	45046	6992
100	89.747	281.63	0.00	5.21	34904	5686
200	94.715	276.66	0.00	5.21	34438	5352
400	355.66	15.72	2.68	2.52	5538	5728
600	365.85	5.53	3.33	1.87	4769	2308
800	367.60	3.77	3.55	1.65	4574	1804
1000	369.07	2.30	3.93	1.27	4408	1172
2000	369.50	1.87	5.09	0.11	4724	1113
3000	369.60	1.77	5.21	0.00	5087	1157

Random trajectory

Figure 17: Results of simulation for different cache sizes. All quantities shown refer to their mean value per frame. The last column is the standard deviation of the number of bits per frame.

all.

References

[CCMS97] A. Ciampalini, P. Cignoni, C. Montani, and R. Scopigno. Multiresolution decimation based on global error. *The Visual Computer*, 13(5):228–246, 1997.

[CMS98] P. Cignoni, C. Montani, and R. Scopigno. A comparison of mesh simplification algorithms. *Computers And Graphics*, 22(1):37–54, 1998.

[DFMMP99a] L. De Floriani, P. Magillo, F. Morando, and E. Puppo. Handling multiresolution meshes on a client-server architecture. Technical Report DISI-TR-99-13, Department of Computer and Information Science, University of Genova (Italy), 1999.

[DFMMP99b] L. De Floriani, P. Magillo, F. Morando, and E. Puppo. A reversible automaton for compressed coding of triangulated polygons. Technical Report DISI-TR-99-16, Department of Computer and Information Science, University of Genova (Italy), 1999.

[DFMP97] 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.

[DFMP98] L. De Floriani, P. Magillo, and E. Puppo. Efficient implementation of multi-triangulations. In *Proceedings IEEE Visualization 98*, pages 43–50, Research Triangle Park, NC (USA), October 1998.

[DFMP99] L. De Floriani, P. Magillo, and E. Puppo. Compressing triangulated irregular networks. *Geoinformatica*, (to appear), 1999.

[DFPM97] L. De Floriani, E. Puppo, and P. Magillo. A formal approach to multiresolution modeling. In R. Klein, W. Straßer, and R. Rau, editors, *Geometric Modeling: Theory and Practice*. Springer-Verlag, 1997.

[Tau98] G. Taubin et al. Progressive forest split compression. In *Computer Graphics (SIGGRAPH '88 Proceedings)*, pages 123–132. ACM Press, 1998.

[ESV99] J. El-Sana and A. Varshney. Generalized view-dependent simplification. *Computer Graphics Forum*, 18(3):C83–C94, 1999.

[Gar99] M. Garland. Multiresolution modeling: Survey & future opportunities. In *Eurographics '99 – State of the Art Reports*, pages 111–131, 1999.

[GH97] M. Garland and P.S. Heckbert. Surface simplification using quadric error metrics. In *Comp. Graph. Proc., Annual Conf. Series (SIGGRAPH '97), ACM Press*, pages 209–216, 1997.

[GTLH99] A. Guéziec, G. Taubin, F. Lazarus, and W. Horn. A framework for streaming geometry in VRML. *IEEE Computer Graphics and Applications*, pages 68–77, April-March 1999.

[HG97] P. Heckbert and M. Garland. Survey of surface simplification algorithms. Technical report, Carnegie Mellon University - Dept. of Computer Science, 1997. (to appear).

[Hop96] H. Hoppe. Progressive meshes. In *ACM Computer Graphics Proceedings, Annual Conference Series, (SIGGRAPH '96)*, pages 99–108, 1996.

[Hop97] H. Hoppe. View-dependent refinement of progressive meshes. In *ACM Computer Graphics Proceedings, Annual Conference Series, (SIGGRAPH '97)*, pages 189–198, 1997.

[Hop98] H. Hoppe. Efficient implementation of progressive meshes. *Computers & Graphics*, 22:27–36, 1998.

[Koe99] R. Koenen. MPEG-4: Multimedia for our time. *IEEE Spectrum*, 36(2):26–33, February 1999.

[PS85] F.P. Preparata and M.I. Shamos. *Computational Geometry - An Introduction.* Springer-Verlag, 1985.

[PS97] E. Puppo and R. Scopigno. Simplification, lod, and multiresolution - principles and applications. In *Eurographics '97 Tutorial Notes*, 1997.

[Pup96] E. Puppo. Variable resolution terrain surfaces. In *Proceedings Eight Canadian Conference on Computational Geometry*, pages 202–210, Ottawa, Canada, August 12-15 1996.

[SvK97] J. Snoeyink and M. van Kreveld. Linear-time reconstruction of Delaunay triangulations with applications. In *Proceedings 5th European Symposium on Algorithms*, 1997.

[SZL92] W.J. Schroeder, J.A. Zarge, and W.E. Lorensen. Decimation of triangle meshes. In Edwin E. Catmull, editor, *ACM Computer Graphics (SIGGRAPH '92 Proceedings)*, volume 26, pages 65–70, July 1992.

[XESV97] J.C. Xia, J. El-Sana, and A. Varshney. Adaptive real-time level-of-detail-based rendering for polygonal models. *IEEE Transactions on Visualization and Computer Graphics*, 3(2):171–183, 1997.

[ZL77] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.