External Memory Management and Simplification of Huge Meshes

P. Cignoni, C. Montani, C. Rocchini, R. Scopigno

Abstract— **Very large triangle meshes, i.e. meshes composed of millions of faces, are becoming common in many applications. Obviously, processing, rendering, transmission and archival of these meshes are not simple tasks. Mesh simplification and LOD management are a rather mature technology that in many cases can efficiently manage complex data. But only few available systems can manage meshes characterized by a huge size: RAM size is often a severe bottleneck.**

In this paper we present a data structure called Octree-based External Memory Mesh (*OEMM*). It supports external memory management of complex meshes, loading dynamically in main memory only the selected sections and preserving data consistency during local updates. The functionalities implemented on this data structure (simplification, detail preservation, mesh editing, visualization and inspection) can be applied to huge triangles meshes on low-cost PC platforms. The time overhead due to the external memory management is affordable. Results of the test of our system on complex meshes are presented.

CR Descriptors: I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling - *Curve, surface, solid and object representation*; **I.3.6 [Computer Graphics]: Methodology and Techniques.**

Additional Keywords: Out-Of-Core Algorithms, Hierarchical Data Structures, Mesh Simplification, Level of Detail, 3D Scanning, Texture Synthesis.

I. Introduction and state of the art

Very large triangle meshes, i.e. meshes composed of many millions of faces, are common in many applications: range scanning, volume data visualization, terrain visualization, etc. For example, huge meshes (up to Giga triangles sizes) can be produced by scanning Cultural Heritage artifacts [3], [16] or by processing large volumetric dataset (e.g. the data produced by the Visible Human project or the DOE ASCII project). Obviously, such complex meshes introduce severe problems in the archival, manipulation, visualization and geometric processing. Huge mesh management encompasses different processing goals:

- Efficient visualization, for selective inspection and presentation (direct raw mesh visualization is inefficient when we want to focus on a small dataset region).
- Mesh editing functionalities, to improve the quality of the data (e.g., 3D scanned meshes need smoothing filters or operators for the triangulation of holes).
- High quality simplification capabilities, to allow the construction of LOD representations (even if alternative representations exist, e.g. based on point-based primitives [24], topology is crucial in a number of applications and triangles are still the standard graphics primitive).
- Finally, applications' specific functionalities can be required (e.g. computing digital measures on the model or supporting data conversion tools for rapid prototyping).

Istituto Scienza e Tecnologie dell'Informazione - CNR (formerly IEI-CNR and CNUCE-CNR), Area della Ricerca CNR, v. G. Moruzzi 1, 56124 Pisa, ITALY. E-mail: {cignoni|montani|rocchini}@iei.pi.cnr.it , r.scopigno@cnuce.cnr.it .

The adoption of an *External Memory* (EM) technique is mandatory whenever we want to process a huge mesh on a limited core memory footprint. The design of EM solutions is a very active research area, and many groups are working on this issue in the graphics community as well. Recent results are: EM isosurfaces fitting [4], EM reconstruction of surfaces from point clouds [2], EM visualization [29], or EM solutions for the simplification of huge meshes [13], [17], [7], [18], [28]. Let us focus on the mesh simplification task.

A. Mesh Simplification

In the context of geometric processing, mesh simplification can be considered a crucial task, and core memory is often the bottleneck [10]. Almost all simplification tools require the whole mesh to be loaded in main memory. If we consider Quadric Error edge-collapse simplification [11], the space complexity can be estimated as a factor of the mesh size (176 byte for each face). Therefore, we can process around 1.1M-1.3M faces on a system with 256MB RAM.

Various techniques have been presented to face the problem of huge mesh simplification: Hoppe's hierarchical method for digital terrain management [13], that can be extended to 3D meshes as shown in [22], [8]; the clustering solution proposed by Lindstrom [17]; and the spanned mesh simplification algorithm by El-Sana et al. [7].

Hoppe hierarchically divides the mesh in blocks, simplifies each block by *collapsing edges* (the collapse of elements incident on the boundary of the block is forbidden) and then traverses bottom-up the hierarchical structure by merging sibling cells and again simplifying. This approach has either a bottleneck on the output size (because a complete bottom-up traversal of the tree is required to remove elements incident on the inter-cell boundaries) or on the simplification accuracy (intermediate results present unpleasant runs of original high resolution elements located on cells boundaries). Moreover, this approach cannot be extended easily to support other geometric processing tasks, because the elements shared by adjacent blocks cannot be modified unless the blocks are merged. These disadvantages are shared with the 3D mesh extensions of the Hoppe's approach [8], [22].

The *clustering* algorithm [23] can be easily implemented in external memory [17] and guarantees excellent time effi-

ciency. Unfortunately, the accuracy of the mesh produced is much lower if compared with the accuracy of methods based on edge collapse. The simple criterion adopted (unify all mesh elements that are contained in the same cluster) implies that every shape feature whose size is smaller than a cluster cell is removed. Clustering performs an *accurate* (it is based on quadric error metrics) but *regular* sub-sampling in the model space. Therefore, it is not able to simplify large sections of the mesh which have a low curvature variation and span multiple cells. A disadvantage of the original clustering approach is that intermediate simplification results are maintained in main memory. This prevents simplification when an intermediate reduction rate is requested. This latter problem has been recently solved by Lindstrom and Silva [18] by storing the output mesh and intermediate data on disk (out-of-core sorting is used to detect and compose the quadrics associated to each grid cell); output size independence is obtained at the expenses of two to five times slower simplification times. Another improvement over the general Clustering approach has been proposed recently by Shaffer and Garland [28]. A higher quality approximation is obtained at the expenses of a small time overhead (around two times slower than standard clustering) by replacing the regular grid with an adaptive subdivision based on BSP trees. The external memory implementation needs multiple scan of the data: initially, a uniform grid is used to quantize the input data and compute quadrics; then, this info allows to build an adaptive subdivision of the space (by a BSP tree), that is used in the last step to simplify the mesh. Even if this method gives an improved accuracy with respect to standard clustering (given a budget of K output vertices, these are positioned on the surface in an adaptive manner), the accuracy is lower than that produced with edge-collapse methods and the output is often non-topologically clean, as well as all clustering solutions (see some discussion in Section VIII).

The spanned mesh simplification algorithm by El-Sana et al. [7] starts from an indexed mesh with explicit topology. It keeps all the edges of the mesh (with the adjacent faces) into an external memory heap, ordered according an error criterion based on edge length (using edge length does not ensure high accuracy in simplification. Implementing an ordering criteria based on quadric error metrics [11] is not easy, due to the more complex data loading required for the initial evaluation of the QEM for each edge and for the update of QEM after each edge collapse. Given the k edges on top of the heap (k depends on the core memory size), it loads in memory the associated adjacent faces pairs and reconstructs the mesh portions spanned by these edges. Then, the edges having all their incident faces loaded in memory can be collapsed. This approach reaches a good computational efficiency if we are able to load in memory a large percentage of the data (i.e. large contiguous regions). In the case of huge meshes the shortest edges could be uniformly scattered and it could happen that most of the spanned sub-meshes loaded in memory consist of only a few triangles, therefore requiring a very frequent loading/unloading of very small regions. A positive advantage of this method is that the simplification order performed by the external memory implementation is exactly identical to the one used by an analogous in-core solution (thus, the mesh produced by the external memory implementation is identical to the one of the in-core solution).

Beside the specific limitation of each one of the above techniques, most of them have been designed to support just simplification, and extending these approaches to support also other geometric processing algorithms can be not straightforward.

B. Objectives

Our goal is therefore to support general huge mesh management on low-cost platforms, by providing mesh manipulation, editing, filtering, simplification and inspection features under the constraint of a limited memory size. None of the existing systems support these features, especially if we consider PC-based systems. Our system is based on a hierarchical data structure, called Octree-based External Memory Mesh (OEMM). Hierarchical schemes [25] have been often used in geometric processing and interactive visualization [1], [8], [9], [13], [15], [22], [24], [29], but in all these cases the hierarchical structure sets strong limitation on how and where processing can be performed. For example, the hierarchical simplification approach [13], [8], [22] simplifies some boundary elements only in the very last step of the bottom-up simplification process (i.e. boundary elements can be managed only when the corresponding leaf nodes are merged). Our external memory structure is not just another space subdivision or data paging scheme. Peculiar characteristics of our approach are: (a) it supports a global indexed representation (built on any huge mesh given in input as a triangle soup); (b) it allows any partial data load/update/write-back operation, by performing an automatic *on the fly* re-indexing of the loaded data portion: in this way, any loaded portion is represented in core memory with indexed lists containing only the loaded vertices and faces. Data subdivision is performed using a standard octree-based regular split; elements spanning adjacent cells are identified in the construction phase, consistent id's are assigned to the corresponding vertices in adjacent nodes (vertex indexing also satisfies the lexical order of the corresponding octree nodes) and, finally, each border element is assigned to a single node of the octree. This allows data loading of any subset of the mesh, which is converted on the fly in a single, consistent mesh indexed on the local subset of vertices. The potential boundary elements contained in the interior of the loaded region can therefore be treated as any other element, while a *tagging strategy* (the peculiar characteristic of our approach) allows easy detection and management of the elements located on the boundary of the current region. This makes simple the design of the external memory version of many geometric algorithms. Therefore, the underlying space decomposition is completely hidden (and managed by the data structure), and coding geometrical algorithms working on data partitions becomes easier.

Thanks to the freedom of accessing any small subset of

the mesh consistently, we can easily implement different mesh processing algorithms on the *OEMM* data structure, such as: *mesh editing* and *selective inspection*; high quality *mesh simplification* (based on the quadric error metric approach [11]); *detail preservation* (based on bump- or rgb-texture resampling, to encode the high frequency detail lost during simplification [5]). The bottleneck on either input and output data size is thus removed.

The paper is organized as follows. Some definitions are introduced in Section II. Then, the *OEMM* hierarchical structure is introduced in Section III. Details on the construction of an *OEMM* representation from a triangle soup (list of faces, not indexed) are given in Section IV. Section V presents the data management procedures (traversal, loading, updating). Section VI describes how to implement external memory mesh simplification. Other mesh processing tasks have been implemented on the *OEMM* data structure, and are briefly described in Section VII. Finally, Sections VIII and IX report results and conclusions.

II. Definitions

Mesh Terminology. A mesh is called *indexed* if all the triangles are encoded by storing a triple of references to their vertices (either with explicit pointers or with integer indices). Conversely, it is called *raw* (or triangle soup) if the triangles are described with a triple of 3D points and sharing of vertices among adjacent triangles is not considered.

Octree Terminology. Given an axis aligned box B containing the dataset, we recursively partition it in eight sub-regions [20]. Sub-regions are numbered according to their relative coordinates in lexicographic order (see Figure 1 for a 2D example), which defines a total ordering between octree leaves according to a DFS visit [25]. Given an octree node n, we denote with B_n the bounding box corresponding to that node. Each bounding box B is identified by two 3D points $B.min, B.max$.
To avoid ambiguities hereafter when we say that a point p is *contained* into a bounding box B we mean that its coordinates are greater than or equal to $B.min$ and less than $B.max$. In this way any point is contained in one and only one *leaf* node of the octree.

III. Octree-based External Memory Mesh

The Octree-based External Memory Mesh data structure (*OEMM*) provides support for the management of generic processing on huge meshes, under the constraint of limited core memory. *OEMM* is based on a hierarchical geometric partition of the dataset with no vertex replication and consistent vertex indexing between leaf nodes which shares a reference to the same vertex. This hierarchy is coupled with an element tagging strategy that permits to manage in a straightforward manner the partial knowledge of geometry and topology (a common situation when only a small portion of the whole mesh is loaded in each instant of time).

A small mesh portion is assigned to each *OEMM* leaf, based on regular hierarchical decomposition. Only the hierarchical structure of the octree is maintained in main memory: each octree leaf holds the external memory address of the corresponding portion of the mesh. An important feature of the *OEMM* is that it maintains a globally indexed representation of the mesh. Therefore, each vertex is uniquely identified by an integer and triangles are described and stored using just three indices (there is no vertex duplication). The vertex indices respect the octree structure and the order defined on the leaves in the following sense: each octree leaf node has associated a unique integer range, and all of its vertex indices lie in this range.

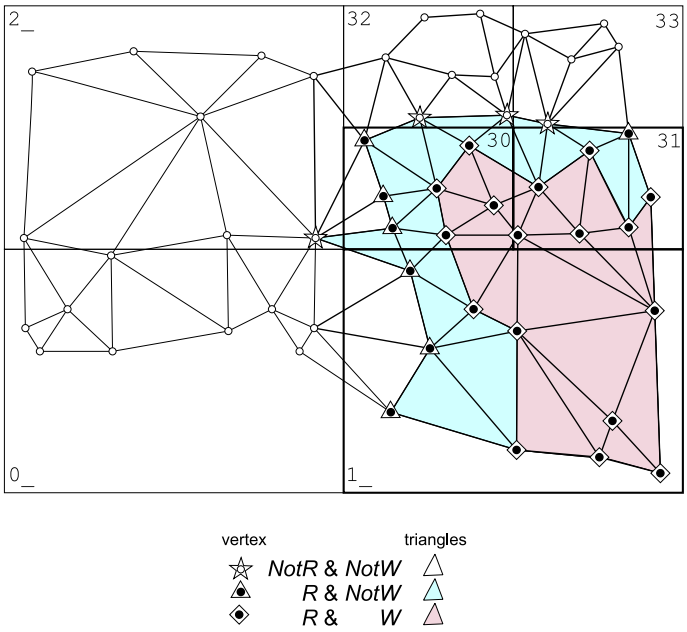


Fig. 1. Flags setting on a section of the mesh currently loaded in main memory (leaf nodes 1, 30, 31 are the ones loaded). Node numbering reflects the lexical order of the nodes.

Definition III.1: **OEMM leaf node.** Each leaf ℓ of the octree stores a pointer to a secondary memory chunk which contains:
• **vertices** - all the vertices **contained** in the bounding box of ℓ ; for each vertex v we also store the indices of the *OEMM* leaf nodes which contain shared faces incident in v;
• **faces** - for each triangle t partially contained in the bounding box of ℓ, t is stored in ℓ only if ℓ is the minimal leaf (according to the lexicographic order) which contains a vertex of t. Therefore, all the triangles completely contained in the bounding box of ℓ are stored in ℓ. In other words, a face is stored in the lowest index leaf that contains a vertex of the triangle.

Maintaining the whole octree structure in main memory is not a memory bottleneck because its memory size is not very large, even for very large meshes. To give an example, if we have an average of 16K triangles in each octree leaf, then the octree structure associated with a 10G faces mesh requires ≈40 MB.
The data structure encoding each *OEMM* node on disk is

as follows:

```
OctreeNode{
  OctreeNode *parent;
  OctreeNode *child[8];
  EM\_Pointer Mesh; // Pointer to external memory
  int vn;           // Vertex number
  int tn;           // Triangle number
  int BaseInd;      // index range of leaf goes from
  int LastInd;      // BaseInd to LastInd
  vector<int> L;    // Set of adjacent leaves with shared data
}
DiskTriangle{
 int v[3];
 data attributes;  // User-defined attributes
}
DiskVertex{
 Coord3d p;
 unsigned char ol[8]; // Indices of adjacent cells containing
                  // triangles incident in vertex p
 bool deleted;
 bool modified;
 data attributes;  // User-defined attributes (color,
                  // quadrics, etc.)
 }
```

This gives a minimal representation of a mesh; if needed, more complex representations (e.g. with explicit topology links between adjacent faces) can be built on the fly at data loading time.

The main purpose of this structure is to allow the user to load in main memory and to modify any small contiguous portion of the mesh, independently of the underlining hierarchical decomposition. By traversing the *OEMM* octree structure and iteratively loading, updating and saving leaves we are able to apply on very large meshes almost any kind of geometric algorithm based on local updates. Moreover, some geometric algorithms that need to work on the whole mesh can be redesigned such that just a portion of the data should be needed at each instant of time.

Loading just a portion of the mesh force us to cope with partial knowledge of the mesh elements. As an example, a vertex on the frontier of the mesh section assigned to the current leaf may have incident faces which are not contained in the current leaf, or some of the faces on the frontier can be defined by vertices whose geometry has not been loaded because it is stored in an adjacent, non-loaded cell. The `ol` field contained in the *vertex* data structure encodes the set of adjacent leaves which contain faces incident in that vertex. To ensure space efficiency, the `ol` has been implemented as a fixed length `unsigned char` field; the values contained in `ol` are indices to a list of adjacent cells stored in the corresponding leaf node, that is the vector `L` of leaf indices (see data structure above). If more than 8 indices are required, they are allocated in a dynamic list (but this situation never arose in all the tests presented). Note that it can be proved that if triangles edges are smaller than half of the smallest octree box, then each vertex can be referenced by, at most, seven other leaves.

Because we load only a portion of the mesh, we must maintain explicit information on which operations can be performed on the currently loaded or referred mesh elements. For this reason a set of flags are added to the `Vertex` and `Triangle` data structure when data are loaded in RAM:

```
Triangle{
vertex* v[3];
int flags;
}
Vertex{
Coord3d p;
int OEMMVertIndex;
int flags;
data attributes;  // User-defined attributes
                  // (color, quadrics, etc.)
}
```

The **vertex flags** hold the following values:
• **readable** and **writable:** a vertex is *readable* if it is contained in one of the currently loaded leaves. A vertex is *writable* if all of the faces incident in it are contained in leaves currently loaded. In this way, a vertex that is referenced by some non-loaded triangle is set *readable* but *non writable*, preventing modifications. On the contrary, if a vertex is not loaded and is referenced by triangles that are loaded then it is tagged as *non readable* and *non writable*. We implicitly assume that *writable* implies *readable*;
• **modified:** a vertex is *modified* when either its coordinates or the set of elements incident in it have been modified or, in some sense, processed (for example, to prevent multiple redundant processing on the same mesh element).

Conversely, the **face flags** hold the following values:
• **readable** and **writable:** a triangle is *readable* if it is contained in one of the currently loaded leaves. A triangle is *writable* if all of the vertex-adjacent triangles are *readable*, or in other words if all its vertices are *writable*;
• **modified:** a triangle is *modified* when its vertex indices have been modified.
An example of flags settings is shown in Figure 1. Flags are initialized by the loading function of the *OEMM* leaves, according to: (a) the values of the `ol` and `L` fields in the *OEMM* representation (see `OctreeNode` and `DiskVertex` data structures), and (b) the current set of leaf nodes loaded.

IV. Building the *OEMM*

We assume that the input mesh comes as a large set of raw, not indexed triangles, stored therefore with just 3D coordinates. We therefore describe OEMM construction considering the worst-case input (if we have in input an indexed mesh, some construction steps described below can be avoided or simplified). In any case, many huge meshes comes as a set of independent indexed meshes (e.g. produced by separate runs of a surface fitting code), and therefore re-indexing them in a common vertex space is needed. The *OEMM* is constructed in two steps:
1. a *raw OEMM* structure is built in secondary memory by processing all input triangles; the *raw OEMM* is a non-indexed *OEMM* . Each *raw OEMM* leaf node ℓ contains *all* triangles $\{t_i\}$ such that at least one of the vertices of t_i is contained in the node bounding box B_ℓ. Note that triangles shared by multiple leaf nodes are replicated in all those nodes of the raw *OEMM* ;
2. the raw *OEMM* is traversed and an *indexed OEMM* is built, i.e. an octree where triangles are indexed us-

ing a global vertex naming strategy. At the end of this phase, vertices and faces of the mesh are partitioned on the *OEMM* leaves according to Definition 3.1, with no redundancy.
In the following paragraph we see some details on how this building process is performed.

A. Building a raw OEMM

This first construction phase is performed in two steps. The goal of the first step is to determine the structure of the *OEMM* octree: we fix a maximal depth of the *OEMM* , we scan all the triangles and count, for each leaf node ℓ of the *OEMM* , how many triangles should be assigned to it. When all faces have been virtually assigned to leaves, sibling leaf nodes are collapsed into the parent node if and only if: the sum of the triangles contained is lower than a user-selected threshold, called *max_triangles*, **and** the resulting merged node has adjacent nodes whose depth in the tree differs from the depth of the current one by no more than three levels (i.e. we build a *restricted* octree [25]). The second condition guarantees that loading a leaf and all of its adjacent leaf nodes has a bounded space complexity.

In the second step we read again the set of raw triangles from secondary storage, and distribute them in the secondary memory buckets corresponding to the octree leaf nodes.

B. Building an indexed OEMM

To build the *indexed OEMM* we perform two complete traversal of the intermediate data structure: firstly, we traverse the raw *OEMM* to build an intermediate indexed *OEMM* where only *internal vertices* are correctly indexed (we call internal the vertices *contained* in the leaf bounding box, and external the others); then, the final indexed *OEMM* is built by indexing also the *external vertices* which belong to the faces shared by the adjacent leaf nodes.

Indexing internal vertices. The indices of the vertices should respect the lexicographic order of the leaves of the *OEMM* . Therefore, the leaves of the *raw OEMM* are read from secondary storage in lexicographic order, and for each leaf ℓ we assign an unique index to each vertex contained in the given leaf, and copy them in the *indexed OEMM*. All the vertices that are not contained in ℓ are indexed with a temporary fake value.
In this step we also setup the per-node and per-vertex list of *OEMM* leaves that contain faces shared with the current leaf node. This can be done easily due to the redundant representation of shared faces in the *raw OEMM*.

Indexing external vertices. The last step computes a correct global index for all the external vertices of the shared faces represented in each leaf. Therefore, each *OEMM* leaf node, with the adjacent ones, is read from secondary memory for the last time, and the global indices assigned to the internal vertices of a cell are propagated to the adjacent ones containing shared triangles as follows:

- all the leaf nodes ℓ_i which share triangles with ℓ are loaded;
- for each vertex $v \notin \ell$ of a shared triangle $t \in \ell$, we replace the fake index initially assigned to v in ℓ with the correct index assigned to v in the leaf node ℓ_j containing v.

V. Working with the OEMM

Working with the *OEMM* involves the iterative application of load/[modify/save] actions onto the *OEMM* leaves. Here we describe the details of these steps.

A. Traversal

In order to apply a geometric algorithm over an *OEMM* we have to define a visiting strategy such that all the vertices and triangles are seen at least once as readable and writable. Loading only a leaf at a time does not allow to get full information on the associated mesh portion and to modify the triangles which are not completely contained in the current leaf. The *OEMM* library implements different atomic data access rules:

- *subtree*: load all the leaves contained in the subtree plus all the leaf nodes adjacent to the nodes of this subtree;
- *bounding-box*: load the minimal set of leaves such that all the vertices contained in the given bounding-box and all the triangles referencing them are loaded.

A geometric algorithm can traverse the *OEMM* choosing any of the previous atomic rules depending on the characteristics of the processing to be performed and on the relative space requirements.

B. Loading Leaves

Loading in main memory a generic set of leaves $S = \{\ell_0, .., \ell_k\}$ means to reconstruct a standard indexed mesh representation from the *OEMM* loaded leaf nodes. This task involves the re-indexing of the mesh faces to a new vertex vector composed only by the loaded vertices (i.e. a vertex vector much smaller than the global *OEMM* vertex list); and to assign the correct flags settings to all faces and vertices.

Vertices re-indexing can be done in linear time because the maximum number of adjacent nodes is bounded by a constant. The original index of each vertex is maintained (see the `int OEMMVertIndex` of the `Vertex` data structure in Section III), in order to guarantee that *non writable* vertices could be placed back in the original position of the corresponding leaf block on secondary memory (see Section V-C).

The flag values (*readable/writable* and *modified*) are assigned as follows (see also Figure 1):

- vertices referenced by triangles outside all $\ell_i \in S$ are tagged *not writable*.
- vertices stored in non-loaded leaves **but** referenced by triangles in $\ell_i \in S$ are replaced with dummy vertices and tagged *not readable, non writable*.

C. Saving Leaves

A modified mesh corresponding to a set of leaf nodes $S = \{\ell_0, .., \ell_k\}$ has to be written back on secondary memory to make these modifications permanent. This step involves a back conversion of the current indexed mesh into a *OEMM* mesh chunk indexed with the global *OEMM* indices. We distribute the vertices to the appropriate *OEMM* leaves, and implicitly assign to each vertex the global index. During the saving step it is important that each vertex referenced by *non loaded* triangles (i.e. the ones that we classify *non-writable*) keeps its original position in the vertex list of the *OEMM* leaf node (the global index of each vertex is implicitly coded with the range of the leaf plus the vertex position inside the leaf).

Then, we distribute triangles to the appropriate *OEMM* leaf; for triangles shared by multiple leaves, the selection is performed by looking at the global index of the vertices, according to definition in III.1. Finally, for each face the indices of its vertices are replaced with the corresponding new global indices.

To ensure correctness of loaded nodes saving back, the following situations must be detected:

- *vertex indices out of range:* if the number of vertices to be saved back in a *OEMM* leaf is bigger than the original leaf range (for example because we updated the leaf to triangulate some mesh holes), then the leaf range should be expanded. Because assigning a wider range to a leaf is a costly operation (involving loading and re-indexing multiple leaves), at *OEMM* creation time we have distributed the leaf ranges uniformly over the 32 bit integer space. In this manner, there is plenty of space between any pair of consecutive leaf range to slightly widen the range. Obviously, if leaf nodes size changes in a drastic manner, an update to the *OEMM* structure could be needed (see Section V-D);
- *vertex coordinates not contained in the current loaded space:* a dangerous situation is when the coordinates of a modified vertex are not contained in the space corresponding to the loaded *OEMM* section (i.e. the union of the bounding box of the loaded leaf nodes). To prevent this situation, we detect every update which modifies the mesh by moving vertices in regions that are still not loaded, abort this update and backtrack.

D. Modifying the OEMM structure

The *OEMM* structure can be dynamically updated due to multiple delete/creation actions operated on the loaded nodes.

Node Merging. Every time a leaf is saved back, we firstly check if it can be collapsed with its siblings nodes in the corresponding parent node. When the number of vertices and triangles of the eight siblings is lower than a given threshold and all the conditions specified in Section IV-A hold, we can merge them in a single leaf. Node merging is as follows: the eight leaves and all other *OEMM* nodes referencing their vertices are loaded; the new range of the vertex indices assigned to the new leaf is computed; vertices are re-indexed; all the triangles of the loaded leaves are remapped with the new vertex indexing (this can involve the updating of some `ol` lists of the adjacent nodes); finally, all the loaded nodes are saved back.

The merging process is executed frequently during external memory mesh simplification (see Section VI).

Node Splitting. Node splitting is the inverse of the previous operation, and it has to be performed when the number of element in a leaf is higher that the maximum leaf size. Again, we have to reindex the vertices of the split sections and to reflect the new vertex indexing on the sibling nodes.

E. OEMM Complexity

While from a theoretic point of view octree's have not a good worst case complexity, they perform really well in practice. Let us assume that the input mesh has some *reasonable* characteristics: the number of triangles incident in a single vertex is bounded by a constant; the size of the faces is not smaller than a minimal value, and therefore the maximal depth of the octree is bounded. Then, we can assert that: loading and saving a leaf node (and some of the adjacent ones) has a cost linear in the size of the mesh elements contained in the loaded/saved nodes.

VI. External Memory Mesh Simplification

Given a triangular mesh we want to reduce its size by adopting a high-quality incremental approach, e.g. based on the iterative collapse of its edges [11]. Locality of the simplification method is a must, to allow us to load and process the mesh one piece at a time. In particular, a Quadric Error Metrics (QEM) method has been implemented in our system, and is described in the next subsection. Each edge collapse has an error-cost that has to be evaluated for each candidate edge, both at initial time and during the simplification process (every time the given edge is adjacent to some modified mesh component). We assume here that the error-cost can be computed in constant time and that requires a *per-vertex* constant space occupation (i.e. it requires only to access a local neighborhood of the collapsed edge). This last assumption is true for the error estimation techniques used in [27], [11], [19]. At each step of the simplification process the edge with the minimal error cost is collapsed (a heap is used to support ordered selection) and the error evaluation of the adjacent edges is updated. The overall worst case complexity of such an algorithm is $O(v \log v)$, with v the number of vertices.

A. Quadric Error Simplification in the OEMM framework

Quadrics are included in the *OEMM* vertex attribute and used to evaluate edge collapse error. As far concerns quadrics management, there are mainly two approaches: storing and updating quadric errors during edge collapse (see Garland and Heckbert [11]) or re-computing quadrics on the fly as proposed in the *memoryless* approach [19]. To describe how do we manage quadrics, we have to distinguish between what we store on disk, and what we store in RAM. Both our RAM-QEM and OEMM-QEM use the approach of Garland-Heckbert (quadrics are saved and updated during the simplification of the currently loaded section of the mesh). In the external-memory implementation

(OEMM-QEM), when the simplification of the current section is terminated we write back on disk just the mesh (and discard the quadrics). Therefore, when the same leaf is loaded again and simplified further, we will start from a set of newly initialized quadrics. Our experience showed that retaining quadrics on RAM (during simplification of a mesh portion) can be worthwhile, while it is not worthwhile to retain them also among different simplification passes over the same section (due to limited impact on accuracy and the substantial overhead on data loading/writing) and it helps to avoid the *quadric lock* problem [14].

Simplification algorithms usually adopt a priority queue to choose the next edge to be collapsed; for this reason they access the mesh with an order that is inherently non-local. This scattering behavior causes *virtual memory trashing*, making any approach based on standard virtual memory features totally inefficient. Instead of forcing the algorithm to follow the exact edge collapse order, as done for example by El-Sana et al. [7], we choose to slightly change the collapse order in order to catch geometric locality. Therefore we do not keep a global heap with all the possible collapses, but we traverse the *OEMM* (following the lexical order of the leaves) and for each subtree that we load we build a local priority queue and simplify it separately. We have verified empirically that this *local sorting* has a very little influence over the quality of the resulting simplification.

For each loaded subtree, we also load all the adjacent leaves of the *OEMM* . This ensures that all the possible edges of the current subtree (including the ones on the boundary of the subtree) are evaluated for a possible collapse. Therefore, at the end of the traversal the mesh is uniformly simplified (while other hierarchical approaches are constrained to leave untouched the inter-cell boundaries [13], [22], [8]). Let ε be the maximum quadric error the mesh should satisfy, we produce a small sequence of errors (ε_1, .., ε_n=ε) built using a logarithmic increasing rule and iterate QEM simplification n times on the mesh. At each iteration i, we visit all the *OEMM* leaf nodes following the *subtree* traversal rule (see Section V-A); QEM is run on each mesh portion as long as accuracy ε_i is satisfied. During QEM run, all the edges that are incident in *writable* triangles are evaluated for collapse, and the corresponding forecasted error is stored in the heap. The use of the *readable* and *writable* flags is defined easily. We can collapse an edge only if all the vertices connected by an edge to any of the edge's vertices are *writable* and all the vertices connected with an edge with these ones are *readable*. This because for the collapse of an edge we need to *modify* (alias writable permission) the vertices at topological distance 1, and to know the value (alias readable permission) of the vertices at topological distance 2 (because we need to know their data to evaluate the new approximation error of all vertices at topological distance 1).
When we have reached error ε_i on a given *OEMM* mesh portion, we check during the leaf saving procedure (described in Subsection V-C) if it is possible to merge any modified leaf with the siblings leaves, and then we proceed with the next mesh portion. When the user requests a drastic simplification, the final *OEMM* can be composed of one or a few nodes. The traversing scheme ensures that all the edges whose edge stars span on adjacent *OEMM* nodes are considered for collapse at least once in each iteration.

A special case has to be considered, that is the case of edges whose extremes are not contained in two adjacent *OEMM* nodes. This situation is not common in the case of 3D scanned dataset (where data resolution is sufficiently regular), but can occur on CAD data or on irregularly shaped meshes where very *long* or *wide* faces might have vertices contained in non-adjacent nodes. In our approach these faces (spanning non-adjacent *OEMM* cells) are simplified only when, after some simplification steps, they become part of adjacent leaf nodes. Because siblings leaf nodes will be automatically merged during simplification, after a number of steps any "long/wide" face will become either contained in a single leaf node or shared by adjacent leaf nodes. One can object that in this manner the order of simplification of these faces is altered with respect to the standard error-driven order of an *in-core* simplifier. This is true, but we should say that normal meshes contain in general just a few of these "critical" faces (not hundreds or thousands), at least if the data producer has used a solid modeler in a conscious way. Under this assumption and because of their relative size and small number, postponing simplification will not have a drastic effect on the output mesh size/accuracy.

The simplification of mesh topology is needed by many applications, especially when the input data are very complex assemblies. Extending our external memory simplifier to support topology simplification could be easy. Following the approach proposed in [11], given the set of loaded *OEMM* leaf nodes we should only build a uniform grid on the corresponding mesh vertices. This grid supports an efficient detection of the pairs of non-adjacent but close vertices which have to be evaluated for collapse.

B. Detail preservation via resampled textures

Preservation of detail is a must on big meshes, especially if we want to process data with a very complex surface texture (see for example Figure 4) or a complex pictorial detail. In this case, the solutions that evaluate in an integrated manner the approximation of both the shape and some other scalar/vetorial field are in general not adequate, at least if we want to obtain a drastic mesh simplification. Preservation of mesh attributes can be managed as a post-processing phase: a texture can be resampled from the original mesh, containing a discretized representation of the detail removed during simplification (color, high-frequency surface perturbations, other scalar/vectorial fields, etc.) [5]. The resampled texture map is then used at rendering time to paint the detail of original high resolution mesh onto the simplified one [5], [26]. This solution is independent of the simplification process and thus we can simplify the mesh by considering only the shape attribute, leading to very high compression ratios.

The *external memory* implementation of the detail preserving approach is very easy on the *OEMM* framework.

Given a simplified mesh S, we distribute S in an *OEMM* octree having the same structure of the original input mesh *OEMM* . Then the two *OEMM* are traversed in parallel, each face of S is sampled by considering the corresponding mesh section of the original mesh (which is currently loaded in RAM) and the corresponding texture chunk is built. Write back of *OEMM* leaf nodes of the original mesh is not needed, because the data encoded in the *OEMM* is not modified during this phase.

An example of a resampled bump-texture mapped on a very simple mesh obtained by simplification is shown in Figure 7.

VII. Other EM mesh processing tasks

The other tools implemented on top of the *OEMM* representation are described briefly here, for the sake of conciseness.

A *mesh editing* tool has been defined, that allows the user to perform many editing actions which are crucial in a number of applications, e.g. 3D scanning and rapid prototyping. The editing operators provided include: topological check of the mesh, detection of non-manifold components, detection of holes, automatic or user assisted hole-triangulation, elimination on request of complex vertices/faces and small components. Implementing these mesh editing operations on the *OEMM* representation scheme is straightforward.

Obviously, visualization is an important task for the evaluation and the inspection of a mesh. A snapshot of the main window of an external memory visualization session is shown in Figure 2. Implementing an *external memory visualizer* is straightforward, because we only have to define an interface which allows the user to select the *OEMM* leaf nodes to be visualized. The visualization features provided in our prototypal system allow to: visualize a huge mesh by showing the bounding box of all mesh portions contained in the *OEMM* leaf nodes; selective visualization of the mesh sections corresponding to some *OEMM* leaf nodes; color-enhanced visualization of mesh components, to differentiate different topological classes of elements (e.g. for easy visualization of the holes or of the complex vertices detected by the mesh editing module); interactive picking of mesh components; etc. The main goal of this tool is not the pure presentation of the data (which could be implemented also by adopting a point-based approach [24], [21]), but the inspection of the geo-topological characteristics of a given high-resolution mesh (e.g. to evaluate its quality and, in case, to apply editing actions).

VIII. Results

Among the *external memory* algorithms presented, the most complex is the mesh simplification one. We report here the results relative to the simplification of four meshes, all of them obtained by 3D scanning and available at the Stanford 3D Scanning Repository (http://www-graphics.stanford.edu/data/3Dscanrep/):

- the *Happy Buddha* mesh (543,652 vertices, 1,087,716 triangles);
- the *S. Matthew* complete model (186,984,410 vertices and 372,767,445 triangles), representing one of Michelangelo's unfinished statues scanned by the Digital Michelangelo Project [16];
- two *David* models reconstructed at 1mm and 2mm accuracy (respectively: 28,184,526 v. 56,230,343 tr., and 4,128,614 v. 8,254,150 tr.), also scanned by the Digital Michelangelo Project.

Input Data	Simplification			
	quadric error	size (tr.)	time	t/sec
S.Matthew	0 → 1e-5	94,116,116	10:57:37	6.8K
	1e-5 → 1e-3	25,280,206	2:30:54	7.4K
	1e-3 → 1e-1	6,138,792	0:37:05	8.4K
	1e-1 → 1	3,119,222	0:07:29	6.5K
	1 → 10	1,638,646	0:03:21	7.1K
	10 → 100	788,202	0:01:29	9.3K
David 1mm	0 → 1e-2	13,525,698	1:02:24	10.8K
	1e-2 → 1e-1	7,565,958	0:12:31	7.7K
	1e-1 → 1	3,682,158	0:06:32	9.6K
	1 → 10	1,723,895	0:03:07	10.2K
David 2mm	0 → 1	2,517,234	0:07:30	12.5K
	1 → 10	1,413,304	0:01:31	11.9K
	10 → 100	739,485	0:00:52	12.6K

TABLE II

Results obtained in the simplification of the sample meshes. Times are in *hh:mm:ss* (I/O times included). The simplification rate is shown in the last colume (t/sec: simplified triangles per second). The RAM used is around 80 MB.

We did not considered typical CAD datasets. Even if very complex datasets are common in CAD applications, they are usually modeled as a composition (either hierarchical or linear) of medium-sized components, which can often be simplified and managed independently using standard in-core techniques.

Some numerical data on *OEMM* construction and mesh simplification are presented in Tables I and II. The computer used for the tests is a PentiumIII 800 MHz, 256 MB RAM, 30 GB disk running MS WinNT.

The size of the *OEMM* representation (in MB) and the time for the data conversion (from triangle soup to *OEMM*) are shown in Table I. As far concerns the size of the octree, we report here some figures relative to the most complex dataset used, the S. Matthew mesh: the *OEMM* is composed of ≈130K nodes, including internal nodes and empty octree leaves, the triangle per leaf threshold is 16K and the maximum depth of the octree is 8. *OEMM* construction takes a time which is approximately equal to mesh simplification time, and thus rather long. But *OEMM* construction is a data preprocessing phase executed only once, in the framework of the standard pipeline for processing a complex scanned mesh: *OEMM* construction, mesh editing (fixing topology, closing holes, smoothing, etc), mesh simplification. The cost of the conversion process is counterbalanced by the locality of the typical geometric computations (e.g. editing or simplification), which become more efficient on the *OEMM* structure and, obviously, require a small memory footprint. As an example, the simplification

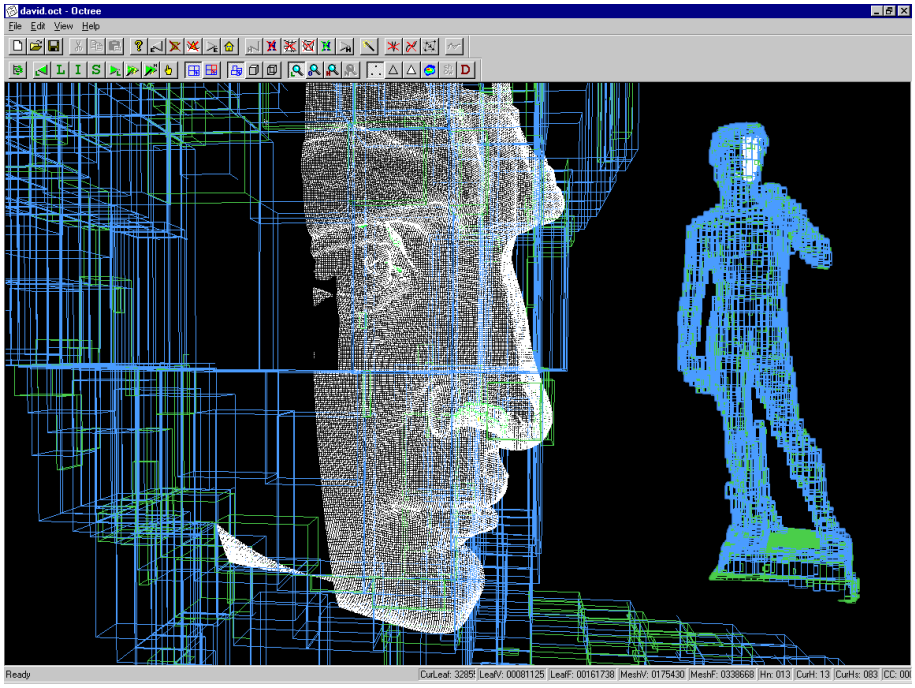
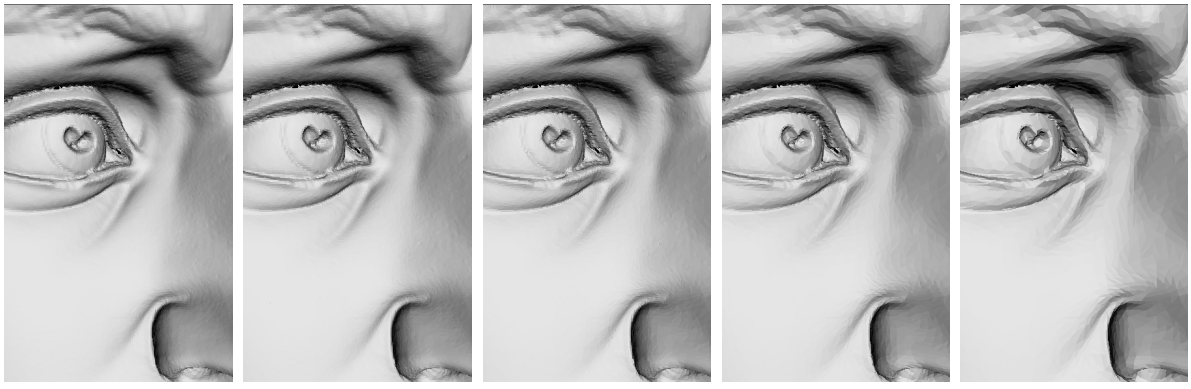


Fig. 2. A snapshot of the main window of the *external memory* visualizer; the loaded mesh section is rendered wireframe, the other *OEMM* leaf nodes are represented by wire-frame bounding boxes.

Input Data			OEMM Repr.			
	mesh size		raw *OEMM*	index.*OEMM*	raw *OEMM*	index.*OEMM*
name	triangles	size	size	size	build time	build time
S.Matthew	372,767,445	7.29 GB	12.5 GB	11.94 GB	2:52:35	8:28:07
David 1mm	56,230,343	1.10 GB	1.85 GB	1.77 GB	0:24:23	1:02:24
David 2mm	8,254,150	166 MB	283 MB	268 MB	0:03:13	0:07:20

TABLE I

THE TABLE REPORTS THE SIZE OF THE TREE SAMPLE MESHES AND OF THE CORRESPONDING *OEMM* REPRESENTATION. TIMES ARE IN *hh:mm:ss* (I/O TIMES INCLUDED).



original data, ~161K tr. (rendered section) | quadr.err.=1e-2, ~39K tr. | quadr.err.=1e-1, ~21K tr. | quadr.err.=1, ~10K tr. | quadr.err.=10, ~6K tr.

Fig. 3. A comparison of the different quality of some simplified David models (1mm David mesh, 53.6 M faces) using the *OEMM* quadric simplification.

of the David and S. Matthew meshes can be performed by using only 80 MB of core memory.

The use of an out-of-core approach introduces some overhead when compared to a standard simplification code working in main memory. We measured empirically the figures of our *OEMM* -based external memory simplifier (OEMM-QEM) with the ones of other codes working in core memory: QSlim v.2, the original QEM implementa-

tion due to M. Garland [12]; RAM-QEM, that is our implementation of the QEM method, running in main memory; and finally our implementation of the OutOfCore Clustering (OOCC) simplifier [17]. Results relative to the Happy Buddha mesh are presented in Table III.

Moreover, one could be interested to know how a standard edge-collapse would perform using just the OS paging mechanism. We run the two *in-core* solutions, QSlim and RAM-QEM, and the *external-memory* OEMM-QEM on a PIII 800 MHz PC with just 128MB of RAM (where no more than 80-90MB are available for user processes); Table IV presents the corresponding running times and global amount of virtual memory[1] (MEM) asked by the process to the OS. The exploding increase of running times when the system starts trashing is evident.

OEMM-QEM and RAM-QEM are based on the same simplification kernel, that is the classical quadric simplification error metric with the addition of weighted factors which take into account the variation of surface normals and the triangle aspect ratio. The difference between OEMM-QEM and RAM-QEM is in the different data traversal and heap management: OEMM-QEM traverses the mesh following the OEMM lexical order, and adopts local heaps to simplify the loaded mesh sections; on the other hand, RAM-QEM uses (analogously to QSlim) a classical global heap and needs to load in memory all of the mesh to initialize the heap and to run simplification. In most cases OEMM-QEM and RAM-QEM produce results (quality and speed) analogous to the ones of QSlim v.2; in some cases, weighting the normal variation improves results accuracy (this is mainly evident in the proximity of discontinue features). Our implementation of the OOCC was as conforming as possible to its original description, including the robust quadric inversion technique described in [17].

The OEMM-QEM consumes around 50% more secondary memory than standard QEM solutions (but secondary memory is nowadays quite an inexpensive resource), but requires a smaller core memory footprint. In any case, consider that the size of the on-disk *OEMM* representation is smaller than the core memory required by an in-core QEM. Therefore, if the core memory is sufficiently large to allow an in-core simplification, it is also sufficiently large to permit the operating system to cache the *OEMM* file in RAM. This explains partially the unexpected results of Table III, where OEMM-QEM simplification time is shorter than the RAM-QEM implementation. Moreover, times are shorter because: the OEMM-QEM local heaps are smaller than the global one used by RAM-QEM (heap construction has complexity $O(n \log n)$); processor cache misses are probably less frequent in the case of OEMM-QEM, because data structure access is more local than that of RAM-QEM.

On larger meshes, the need to perform multiple passes on the dataset (to improve the quality of the simplified mesh, as in the runs reported in Table II) would require multiple loading of the intermediate *OEMM* representations from secondary memory, introducing some overhead with respect to an ideal in-core solution. In fact, the simplification rates reported in the Tables II and III degrade gracefully with the increase of the size of the input mesh. The accuracy of the simplified meshes has been evaluated by using the *Metro* tool [6]. The RMS error (measured as a percentage of the mesh bounding box) is shown in the rightmost columnn of Table III. It is worth to note that *OEMM-QEM* accuracy is slightly lower than our in-core *RAM-QEM*, but at the same time it is still slightly better than the one of *Q-Slim* and obviously much better than *OOCC*.

Happy Buddha (1,087,716 faces)					
	simpl. faces	RAM	time (sec.)	t/sec rate	RMS err
QSlim v.2.0	18,338	195 MB	60	17.4K	0.0131%
RAM-QEM	18,338	160 MB	58	18K	0.0125%
OEMM-QEM					
build (pre-proc)	–	4 MB	58	–	–
simplify	18,338	60 MB	48	21.7K	0.0129%
OOCC	19,071	36 MB	15	69.5K	0.0245%

TABLE III

RESULTS OBTAINED IN THE SIMPLIFICATION OF THE HAPPY BUDDHA MESH, USING FOUR DIFFERENT SIMPLIFICATION CODES.

The simplified meshes produced are shown in Figures 3, 4, 5, and 6.

We performed an empirical comparison with the Out-Of-Core Clustering approach (*OOCC*) [17]. The times of the *OOCC* solution are obviously impressive (see the simplification rate in Table III). On the other hand, the quality of the mesh produced is directly dependent of the regular sub-sampling operated on the mesh (to reach a drastic simplification of a 3D scanned mesh the cluster cell size is generally set much larger than the mean face size). The higher accuracy of the results produced by *OEMM-QEM* is shown in the images presented in Figure 6. Moreover, the meshes produced by the Clustering approach are often non-manifold, and this may introduce problems when we have to apply geometric processing on the output mesh. For example, an *OOCC* run on the 2mm David mesh (from 8M triangles down to 235K) generates more than 21K non-manifold vertices.

One can ask if the improved accuracy of *OEMM-QEM* is worth the processing overhead (*OEMM-QEM* is approximately 3 times slower than *OOCC*). There are a number of applications where data accuracy is a must (visual inspection, rapid prototyping, shape recognition, 3D reconstruction from multiple fragments, etc). In all these cases, a slightly slower simplification time is not a problem: this process is executed only once, and in any case simplification time is a very small fraction of the time needed to produce the raw data (e.g. by 3D scanning) or to analyze it.

A comparison of the different visual accuracy provided by a plain simplified mesh or by the same mesh enhanced

[1]Please note that in Table IV we reported the *working set* used by the simplification process, while the effective maximum size of required RAM has been presented in Table III.

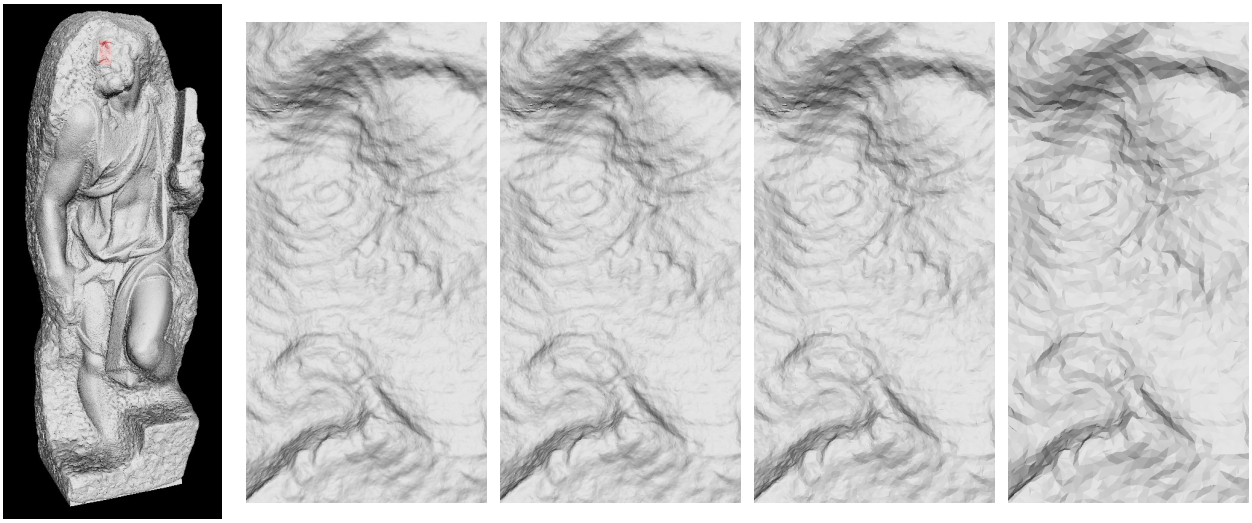


Fig. 4. A simplified model of the S. Matthew statue is shown on the left; a small section of the mesh (S. Matthew's eye and part of the nose) is shown on the right at different accuracies.

HappyBudda (various mesh sizes)										
Input faces:	339,344		408,090		511,138		593,544		746,834	
Output faces:	32,760									
	time (h:m:s)	MEM (MB)	time (h:m:s)	MEM (MB)	time (h:m:s)	MEM (MB)	time (h:m:s)	MEM (MB)	time (h:m:s)	MEM (MB)
QSlim v.2.0	0:00:15	76	0:00:18	90	0:01:46	115	3:17:28	200	n.a.	n.a.
RAM-QEM	0:00:14	94	0:00:16	94	0:00:49	130	0:00:55	180	0:37:19	200
OEMM-QEM	0:00:19	50	0:00:23	50	0:00:29	50	0:00:36	50	0:00:49	50

TABLE IV

The results presented show the poor performances of the in-core solutions (QSlim, RAM-QEM) when the external memory management is demanded to the standard OS paging system; the meshes used in input are simplified versions of the original Happy Buddha.

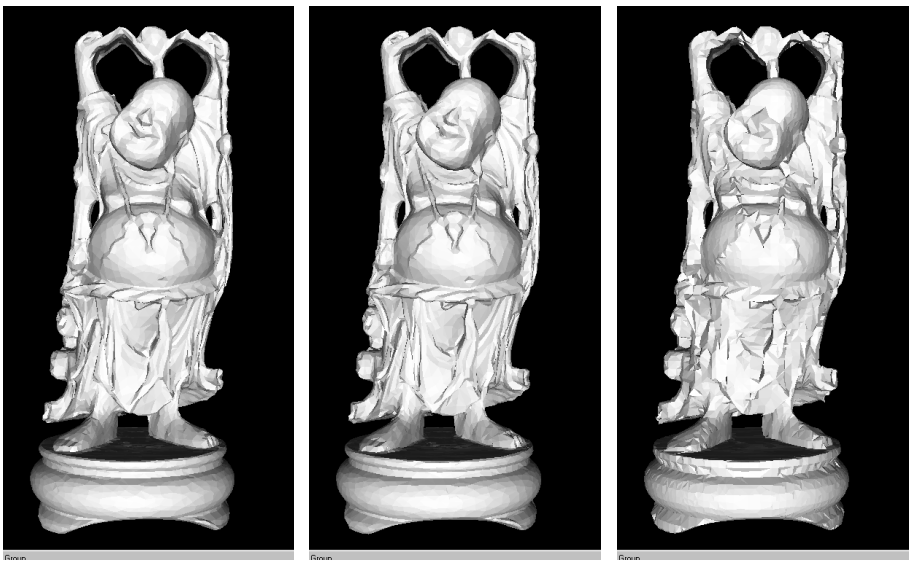
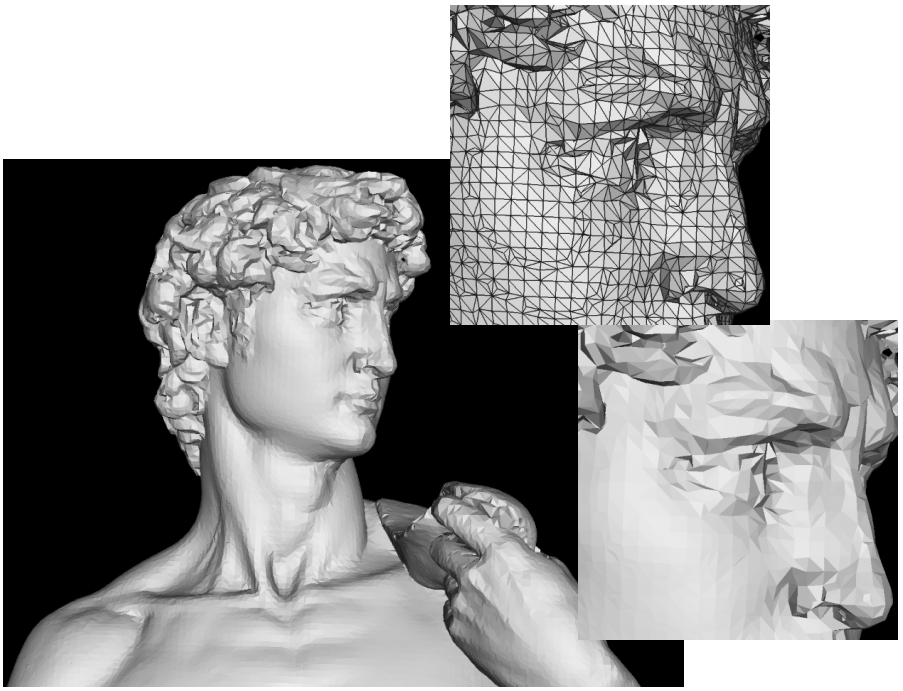
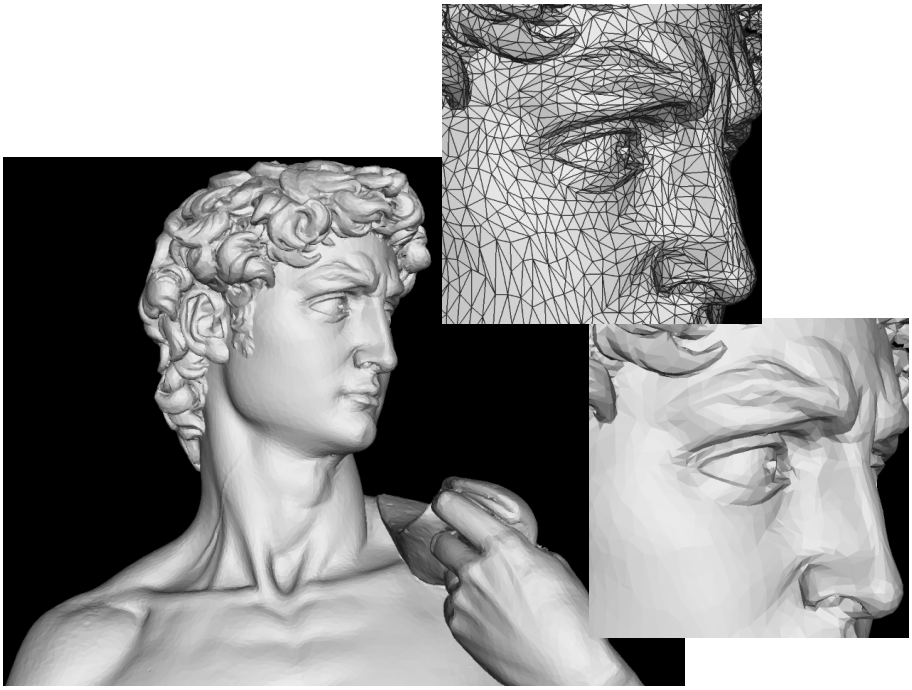


Fig. 5. Results of the simplification of the Happy Buddha mesh.



OOC Clustering, 235K faces



OEMM edge collapse, 215K faces

Fig. 6. A comparison of the different quality of the models obtained from the simplification of the 2mm David mesh (8M faces) using the OOC Clustering solution (top) and the *OEMM* simplifier (bottom).

with a resampled bump-map is shown in Figure 7. Notice how much the visual quality of the drastically simplified mesh (10K faces) is improved by the resampled bump-texture; it appears very similar to a more complex model (1,683K faces) presented in the same image on the left. Considering data size: the 1024*1024 RGB normal map size is 1.5MB, when compressed using PNG format and preserving image quality, and it is texture-mapped to the 10K faces model (size on disk 905KB in un-compressed binary format). This should be compared with the 1.6M

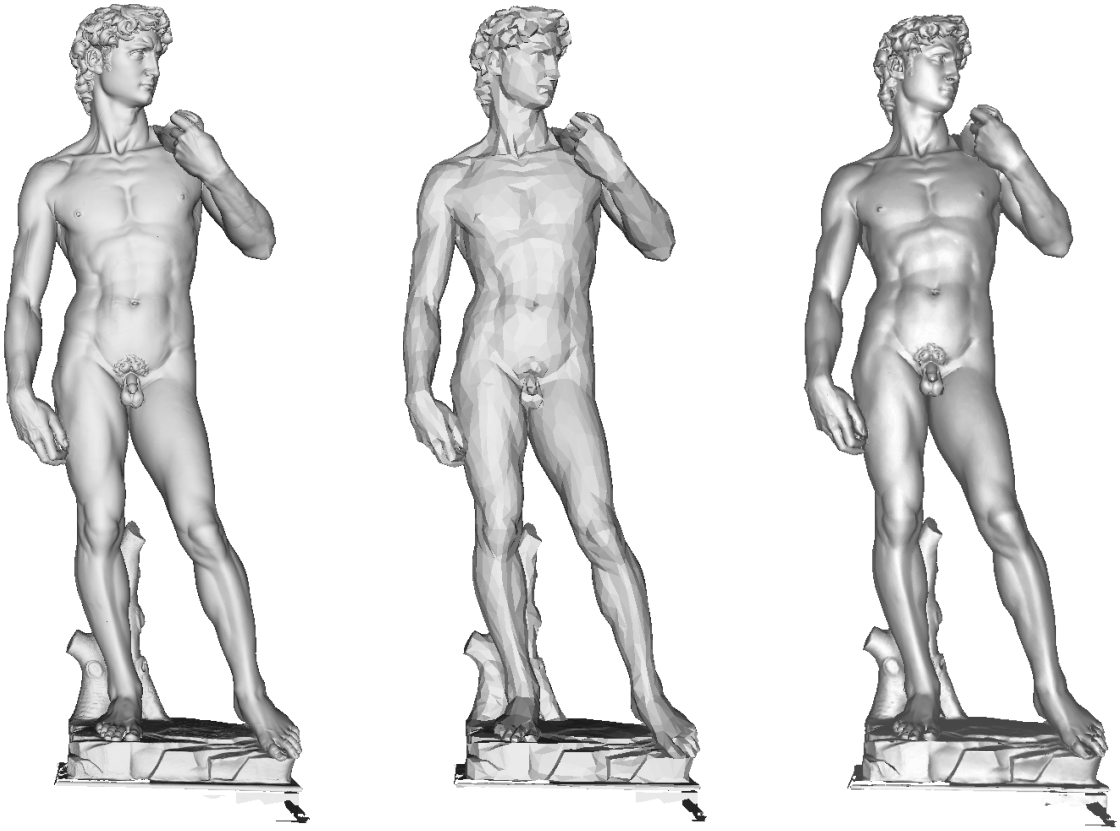


Fig. 7. A comparison of the different visual quality provided by: two simplified David meshes, 1,683K and 10K faces, and the latter enhanced by mapping a re-sampled bump texture (please note that the image with bump-mapping has been created with a different viewer).

faces mesh, which needs 36MB to be stored on disk.

IX. Concluding Remarks

We have demonstrated that even huge meshes can be successfully managed on a low cost architecture. The *OEMM*, an external memory data structure, is at the base of our mesh management and simplification system. It permits to implement in a memory-efficient manner all geometric algorithms that process the mesh via a local update approach, by decoupling mesh size from main memory size and dynamically loading portions of the dataset from secondary memory. The *OEMM* data structure implements an out-of-core global indexed representation on huge meshes, and loading/processing of portions of the data is easy thanks to: the *space subdivision* embedded in the octree representation, the *automatic re-indexing* of the loaded data sections, and the *tagging strategy* (readable/writable tags) that allows the easy detection and management of the elements located on the boundary of the current region. The system presented provides a valid solution for visual inspection, editing, and simplification of huge meshes. As an example, it permits to manage all the post-acquisition phases of the 3D scanning pipeline on a low cost machine. With an acceptable time overhead we can process meshes which cannot be managed on most other architectures. Managing the S. Matthew mesh with an in-core simplifier, for example, would require approximately more than 55GB of core memory. Moreover, an out-of-core solution usually requires a much smaller RAM size than the corresponding RAM-based solutions (in our system, the size of the surface sections loaded can be decided by the user). This appears clear in the results presented in Table III: the simplification of a medium complexity mesh (around 1M faces) works in only 60MB of RAM (or even on a smaller footprint, depending on the size of the loaded subtree selected by the user). Conversely, the RAM-based QSlim solution allocates 195MB to process the same mesh.

It should be noted that the choice of an octree as a partitioning scheme is *not* mandatory. Depending on the mesh processing tasks that have to be carried out, other mesh partitioning schemes can be chosen. For example, if we consider uniformly sampled meshes and tasks that do not drastically alter the size of the mesh (like smoothing filters or hole filling), the octree can be replaced by a simpler uniform grid partition. In this case the interface between the mesh processing algorithm and the *OEMM* remains the same, because it is based on a generic traversal process and the element tagging policy (read/write/modified tags) supported can be easily extended to other decomposition rules.

Possible extensions to the OEMM-based mesh management environment are as follows. We are adding more sophisticated visualization features, which should allow a naive user to navigate and inspect very complex dataset, e.g. meshes produced by 3D scanning Cultural Heritage artefacts, on low cost computers using an LOD approach. We are designing an external memory multiresolution representation, and finally we are planning to include on-the-fly mesh compression techniques to reduce the storage of the *OEMM* leaf nodes.

X. Acknowledgements

We would like to thank Marc Levoy and the Stanford Computer Graphics Group for providing scanned data, and for choosing our simplified meshes as the official simplified models distributed on the project's web.
We acknowledge the financial support of the Progetto "RIS+" of the Tuscany Regional Government and of the EU project IST-2000-28095 "The Virtual Planet".

References

[1] D. Aliaga, J. Cohen, A. Wilson, E. Baker, H. Zhang, C. Erikson, K. Hoff, T. Hudson, W. Stürzlinger, R. Bastos, M. Whitton, F. Brooks, and D. Manocha. MMR: An interactive massive model rendering system using geometric and image-based acceleration. In *1999 Symposium on Interactive 3D Graphics*, pages 199–206, New York, Apr. 26–28 1999. ACM Press.

[2] F. Bernardini, J. Mittleman, H. Rushmeier, C. Silva, and G. Taubin. The ball-pivoting algorithm for surface reconstruction. *IEEE Transactions on Visualization and Computer Graphics*, 5(4):349–359, Oct.-Dec. 1999.

[3] F. Bernardini, J. Mittleman, H. Rushmeier, and G. Taubin. Case study: Scanning Michelangelo's Florentine Pieta'. In *ACM SIGGRAPH 99 Course Notes, Course 8*, August 1999.

[4] Y. Chiang, C. T. Silva, and W.J. Schroeder. Interactive out-of-core isosurface extraction. In *IEEE Visualization '98 Proceedings*, pages 167–175. IEEE Press, 1998.

[5] P. Cignoni, C. Montani, C. Rocchini, R. Scopigno, and M. Tarini. Preserving attribute values on simplified meshes by re-sampling detail textures. *The Visual Computer*, 15(10):519–539, 1999. (preliminary results appeared in IEEE Visualization '98 Proceedings).

[6] P. Cignoni, C. Rocchini, and R. Scopigno. Metro: measuring error on simplified surfaces. *Computer Graphics Forum*, 17(2):167–174, June 1998.

[7] J. El-Sana and Y.-J. Chiang. External memory view-dependent simplification. *Computer Graphics Forum*, 19(3):139–150, August 2000.

[8] Carl M. Erikson. *Hierarchical Levels Of Detail To Accelerate The Rendering Of Large Static And Dynamic Polygonal Environments*. PhD thesis, Department of Computer Science, University of North Carolina at Chapel Hill, 2000.

[9] T.A. Funkhouser. *Database and Display Algorithms for Interactive Visualization of Architectural Models*. PhD thesis, CS Division, UC Berkeley, 1993.

[10] M. Garland. Multiresolution modeling: Survey & future opportunities. In *EUROGRAPHICS'99, State of the Art Report (STAR)*. Eurographics Association, Aire-la-Ville (CH), 1999.

[11] M. Garland and P.S. Heckbert. Surface simplification using quadric error metrics. In *SIGGRAPH 97 Conference Proceedings*, Annual Conference Series, pages 209–216. Addison Wesley, August 1997.

[12] M. Garland and P.S. Heckbert. *QSlim v.2 Simplification Software*. School of Computer Sciences, Carnegie Mellon University, URL: http://www.cs.cmu.edu/ğarland/quadrics/qslim.html, 1999.

[13] H. Hoppe. Smooth view-dependent level-of-detail control and its aplications to terrain rendering. In *IEEE Visualization '98 Conf.*, pages 35–42, 1998.

[14] H. Hoppe. New quadric metric for simplifying meshes with appearance attributes. In *Proceedings of the 10th Annual IEEE Conference on Visualization (VIS-99)*, pages pages 59–66, New York, October 25–28 1999. ACM Press.

[15] D. Laur and P. Hanrahan. Hierarchical splatting: A progressive refinement algorithm for volume rendering. In *Computer Graphics 25(4) (SIGGRAPH 91 Proceedings)*, pages 285–288, July 1991.

[16] M. Levoy, K. Pulli, B. Curless, S. Rusinkiewicz, D. Koller, L. Pereira, M. Ginzton, S. Anderson, J. Davis, J. Ginsberg, J. Shade, and D. Fulk. The Digital Michelangelo Project: 3D scanning of large statues. In *SIGGRAPH 2000, Computer Graphics Proceedings*, Annual Conference Series, pages 131–144. Addison Wesley, July 24-28 2000.

[17] P. Lindstrom. Out-of-core simplification of large polygonal models. In *Comp. Graph. Proc., Annual Conf. Series (SIGGRAPH 2000), ACM Press*, pages 259–262. Addison Wesley, July 22-28 2000.

[18] P. Lindstrom and C.T. Silva. A memory insensitive technique for large model simplification. In *Proc. IEEE Visualization 2001*, pages 121–126. IEEE Press, October 2001.

[19] P. Lindstrom and G. Turk. Evaluation of memoryless simplification. *IEEE Transactions on Visualization and Computer Graphics*, 5(2), April 1999.

[20] D. Meagher. Geometric modeling using octree encoding. *Computer Graphics and Image Processing*, 19(2):129–147, 1982.

[21] H. Pfister, M. Zwicker, J. van Baar, and M. Gross. Surfels: Surface elements as rendering primitives. In Kurt Akeley, editor, *SIGGRAPH 2000, Computer Graphics Proceedings*, Annual Conference Series, pages 335–342. ACM Press - Addison Wesley Longman, 2000.

[22] Chris Prince. Progressive meshes for large models of arbitrary topology. Master's thesis, Department of Computer Science and Engineering, University of Washington, Seattle, August 2000.

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

[24] S. Rusinkiewicz and M. Levoy. QSplat: A multiresolution point rendering system for large meshes. In *Comp. Graph. Proc., Annual Conf. Series (SIGGRAPH 00)*, pages 343–352. ACM Press, July 24-28 2000.

[25] H. Samet. *The design and Analysis of Spatial Data Structures*. Addison Wesley, Reading, MA, 1990.

[26] Pedro V. Sander, Xianfeng Gu, Steven J. Gortler, Hugues Hoppe, and John Snyder. Silhouette clipping. In *SIGGRAPH 2000, Computer Graphics Proceedings*, Annual Conference Series, pages 327–334. Addison Wesley, 2000.

[27] 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.

[28] E. Shaffer and M. Garland. Efficient adaptive simplification of massive meshes. In *Proc. IEEE Visualization 2001*, pages 127–134. IEEE Press, October 2001.

[29] Shyh-Kuang Ueng, Christopher Sikorski, and Kwan-Liu Ma. Out-of-Core Streamline Visualization on Large Unstructured Meshes. *IEEE Tran. on Visualization & Computer Graphics*, 3(4):370–380, October 1997.