OCME: Out-of-Core Mesh Editing Made Practical

Fabio Ganovelli, Roberto Scopigno

***Abstract*—OCME (Out-of-Core Mesh Editing) is a novel data-structure and related algorithms for out-of-core editing of large meshes. OCME uses a hashed multigrid where the triangles are inserted on the base of their size and position. This choice allows a rapid access and, on average, a constant construction time per triangle. Unlike previous approaches, no explicit hierarchy is maintained and therefore insertion/modification/deletion of data does not require costly refitting procedures. OCME stores attributes locally, for example it allows to assign vertex color only to a small subparts of the dataset, and naturally handles multiple-scale datasets.**

***Index Terms*—Out-of-Core Mesh Editing, 3D scanning, Multiresolution Techniques**

I. Introduction

Recent years have seen a surge in the availability of 3D data of the real world obtained with 3D scanning devices. Medium or small size objects can be acquired by triangulation scanners based either on laser or structured lights; medium or large scenes my be targeted by passive methods based on photogrammetryc techniques and time-of-flight or phase-shift scanners (for a survey on these technologies, which is beyond the scope of this paper, the interested reader may refer to [2]). As a result, several digitization projects are producing large 3D datasets with data accuracy and density which were unthinkable until few years ago.

While, over the years, the scientific community has successfully tackled the problems related to the acquisition and processing of 3D data, the current scenario poses new additional difficulties. First of all, the datasets may contain data at different scales, meaning that we may want to edit a scene containing the model of a statue scanned with sub-millimeter precision together with a model of a building scanned at one point per centimeter, or even modeled with a CAD tool. Secondly, the data are noisy and spurious, i.e. we cannot assume to deal with a non-degenerate manifold tessellated surfaces, which is often the final goal of the processing. Instead, datasets are made of points or triangle soups, with all the topological noise usually produced by current processing tools on sampled data. Finally, they can be enriched locally by other attributes such as color, normal, texture coordinates and so on.

OCME is designed keeping in mind these new challenges and offers several advantages over the state of the art:

- it can be updated efficiently, in a time comparable with the pure time for loading the data from secondary memory, and scales well with the size of the dataset;
- it is built on the principle of incremental updates of a initially empty dataset, i.e. there is no *once for all* preprocessing of data.
- it supports the definition of *local* set of attributes such as normal, color and so on.

The design of OCME follows the direct requests of many users of 3D sampling technologies, that urge a software solution for doing elementary editing and cleaning actions over large datasets. Even easy actions (removing vegetations, tourists or pigeons from sampled urban datasets) become an infeasible task if the dataset must be subdivided, processed and finally recombined (which by itself may be a cause for other artifacts to repair).

State-of-the-art approaches to large datasets provide solutions for rendering, compressing, traversal and random access of triangle meshes but fall short in other aspects, the most critical of which is the interactive modification of the dataset (see inset for details).

The typical pipeline is to read a large triangle mesh and to perform a possibly time consuming processing to output a data structure which is more efficient for the some specific goal (e.g. for rendering). For example, many of the existent solutions, more specifically those using space decomposition, first compute the bounding box of the object, upon which the vertex positions are quantized and/or a hierarchy is built. This is done in the assumption that the bounding box is immutable or that the whole data structure can efficiently reflect small changes of it. Unfortunately, if we want a system supporting real dynamic editing, the invariance of the bounding box is wishful thinking, because we want to be able to add/delete surfaces to the dataset, specifically during a massive 3D scanning campaign where data are coming piece by piece and need to be assembled and edited incrementally. Just to make a specific practical example, sampled dataset usually contain spurious data that have to be deleted; many of these are noisy sampled located on the periphery of the model, whose deletion will severely affect the bounding box extent.

Another assumption commonly made is that the meshes are almost a single manifold component, so that they can be (possibly recursively) subdivided in *charts* whose border may be exploited for more efficient compression. Again, sampled data are well away from this situation.

Finally, most of the work is concentrated in a single type of data, while modern digitization devices and algorithms may output triangles, points, colors or, more in general, surface reflection values and we can easily have the case where different parts of the dataset have different attributes.

II. The OCME Data Structure

The key observation behind our approach is that 3D scanning data are, at least locally, quite uniformly dense (since triangles shape vary according to the slopes of the sampled

Visual Computing Lab, ISTI-CNR
email:name.surname@isti.cnr.it

RELATED WORK

In the following we briefly overview the vast literature on handling massive meshes, making a distinction on what "handling" stands for: compressing and accessessing or rendering. For an extensive analysis on these topics the reader my refer to [7] and [8] respectively.

Compressing and Accessing Massive Models

Early techniques for mesh compression were concerned with two aspects: minimizing the number of bits to store the mesh connectivity, by encoding the triangles of the meshes so that *triangles to vertex* references could be encoded efficiently (see for example the Edgebreaker algorithm by Rossignac and the topological surgery by Taubin); reducing the cost of encoding the vertex geometry by quantization and/or predictive coding. These methods usually assume that the mesh could entirely fit in memory. As meshes became too large, a number of techniques adopted the idea of *partitioning in smaller chunks* that could individually loaded and processed. A different approach to massive meshes focuses more on efficient access to the data than on compression/simplification, i.e. the goal is to minimize the number of cache misses when accessing the data, by rearranging the data using space filling curves, so that nearby elements in three dimensional space are also probably close in the external memory (see for example the work by Pascucci and Lindstrom). Producing optimal locality over the data is also the goal of the *streaming* approaches (see the work by Isenburg), where triangle-vertices connectivity is mapped to nearby positions in the data file so that the whole mesh can be traversed with a limited in-core memory. These approaches are more suitable for pipelined algorithms, more than for giving direct access to the model. More recently, mesh partitioning and layouts have been combined in a framework providing both compression and direct access to the data [9].
For the specific case of point clouds, for which things are easier, solutions for interactive editing have been proposed using non-uniform grids combined with a cluster-like impostor technique [1] or using an oct-tree where points are quantized within each node of the oct-tree down to the leaves [10].

Rendering Massive Models

Since that for massive models it is unfeasible to simply load the data from the disk and send them down to the rendering pipeline at the required pace (20-25 fps), a vast number of techniques use intermediate "simplified" representations to trade efficiency for speed, building over the concept of *impostor*: a representation that looks like the original data when its projection on the screen is under a certain number of pixels, but which is more efficient to transmit/render than the corresponding original data. These algorithms come in many flavors but essentially all of them adopt a hierarchy of representations for which only the node indexes are kept in main memory; this hierarchy is visited at each frame for finding the more appropriate set of representations to be used to render the current view, which are concurrently fetched from disk and sent to the GPU.
These methods allow to render hundred millions triangles models and scale well with modern GPU architectures; they work on the assumption that the model is finalized, therefore the data structures they use do not allow editing of the original data, neither they give a way to edit a portion of the dataset and to update the corresponding data structure without resorting to recompute the whole data structure (although it seems possible in principle).

surfaces w.r.t.. the sampling direction), therefore triangles sharing vertices tend to have similar size.
If we had to define a data structure for handling a single large uniform mesh, we could safely choose a regular grid covering the mesh's bounding box and partitioning the volume so that each non empty cell stores roughly the same number of triangles, like in [6]. The advantages of a regular partition scheme are that the per-primitive workload to build the data structure is minimal and the resulting chunks can be compressed and transferred between in-core and out-of-core memory efficiently. We are quite far from this situation because we want to allow free editing and therefore no *a priori* knowledge is given neither about the bounding box of the scene nor about the average size of the primitives.

A. Hashed Multigrid Representation

To retain the advantages of a regular partition without assumptions about quality of data we adopt a *uniform multigrid*, i.e. a hierarchy of regular grids where only non empty cells are explicitly stored and accessed through a hash table. We address the cells of the multigrid structure by a four-integers code (i, j, k, h), which indicates the cubic cell whose lower corner is $(i \cdot 2^h, j \cdot 2^h, k \cdot 2^h)$ and whose size is 2^h. The triangles are assigned to the cells on the base of their $3D$ *position* and *size*. The position of a triangle can be taken either as the position of its barycenter or as one of its vertices while its size can be the length of the longest edge or the area of the triangle, although other choices are possible for both quantities. The idea is that the bigger is the triangle, the higher is the grid level to which it is assigned (i.e. big triangles in big cells). If the mesh is the sampling of

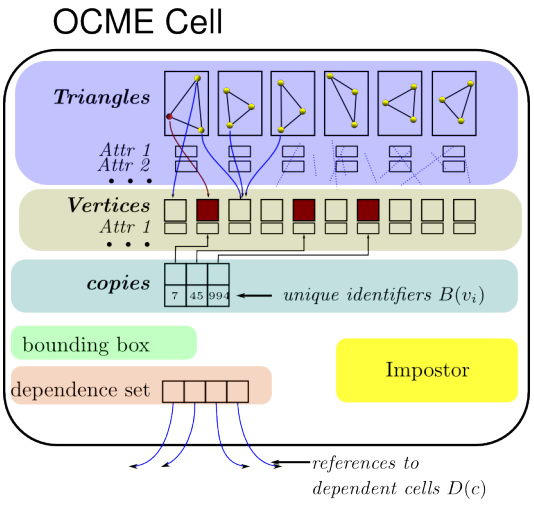

Fig. 1. A scheme of the data contained in a cell.

a real object, then most of it it will be assigned to the same level of the multigrid and each cell will roughly contain a user-defined limited number of triangles.
In order to perform local changes to the model, we could expose access and modification methods to the single elements (face, vertices etc.). Although providing such a transparent access to a large mesh is, in principle, appealing and elegant, we must ask ourselves if and when this is really useful. In our opinion, the main reason for requiring a transparent access is to make it easy to reuse existing code developed for the in-core case, maybe using some existent geometric processing libraries such as CGAL or OpenMesh. However, left aside trivial local computations such as smoothing, these algorithms will use some temporary data structure whose size will likely be proportional to the mesh being processed. Obvious examples are the edge collapse decimation, where we need the stack of possible collapses, or mesh parameterization, for which ad hoc out-of-core solutions are being used. Instead OCME follows an Add/Edit/Commit paradigm: we allow the user to navigate the dataset with a system of impostors, to select the required portion of the dataset and to load it in-core, such that it can be processed in-core and recommitted once finished to the out-of-core data structure.

B. Handling borders

Figure 1 shows the data structure containing the information for a single cell. Each cell stores a complete subset of the mesh, i.e. a set of triangles and all the vertices referred by them, which means that vertices referred by triangles contained in more than one cell are duplicated. We refer to these vertices as *border vertices* and call their local instances *copies*.
Each border vertex v_i is uniquely identified by a progressive number $B(v_i)$, which is stored with all its copies, so that when we want to build an in-core mesh from a group of cells we simply collapse all the vertices with the same $B(vi)$. Figure 2 illustrates the simple example of a mesh partitioned into 4 cells and the assignment of B to the copies (the numbers inside the squares).

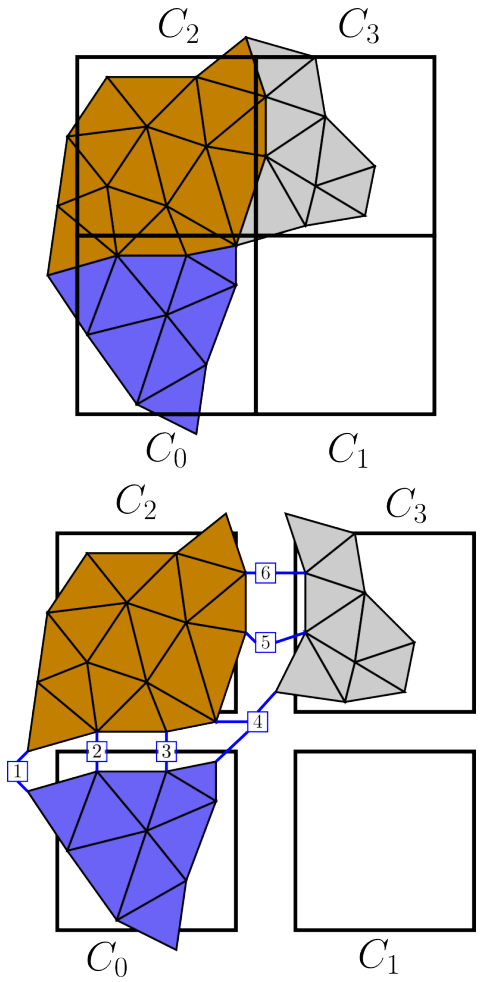

Fig. 2. Handling the borders.

If two cells store copies of the same border vertex we say that they are *dependent* on each other, because if we edit the content of one we possibly affect the content of the other. In the example of Figure 2, if we edit the content of cell c_0, we must take into account that we are also editing the copies of the border vertices 1,2,3 and 4 contained in c_2 and c_3. The data structure to store the the values of B consists in a list of pairs $(vi, B(vi))$ where vi is a local reference to a vertex (see *copies* in Figure 1).
In OCME, when we load a cell c for editing, we create an in-core mesh with the content of $c \cup D(c)$, where $D(c)$ is the set of cells that are dependent on c, called *dependence set* from now on. Then, only the content of c will be actually being editable, while the content of cells in $D(c)$ will be "locked", except for the border vertices shared with c. The concept of dependency is easily extended to set of cells as: $D(c_0, \ldots, c_n) = \cup D(c_i)$, $i = 1 \ldots n$.

III. Add/Edit/Commit in OCME

A. Inserting new data

Figure 3 shows the algorithm for inserting a set of triangles in OCME. At line 3 the *scale* of the triangles is computed on the base of triangle size and then, at line 4, the cell in the

```
1 Algorithm AddMesh(mesh) {
2  for all t in mesh.triangles{
3   h = ComputeScale(t)
4   c = ComputeCell(t,h);
5   AddToGridCell(t,c);
6   CreateCopies();
7 }}
```

Fig. 3. Algorithm for inserting a mesh in OCME.

```
1  Algorithm AddToGridCell(t,c)
2   for(int i = 0; i < 3; ++i){
3    Vertex v = t.vertex[i];
4    int id = GAddr[v].IndexIn(c);  // index of v in cell c
5    if(id != -1)                   // -1 == not found
6     SetReference(t,i,id);
7    else
8     GAddr[v] += AddVertex(c,v);  // add v to c
9    }
10 Algorithm CreateCopies( ){
11   for all v in mesh.vertices
12    if(GAddr[v].size() > 1){
13      nB = nB + 1; // global counter of border vertices
14      for all c in GAddr[v]
15        AddCopy(c,GAddr[v].IndexIn(c),nB);
16   }}
```

Fig. 4. Algorithms for inserting a triangle in a cell and for creating *copies*

proper level is computed on the base of the triangle position. Finally, at line 5, the triangle is actually added to the proper cell and entries are created for all the border vertices.

ComputeScale and ComputeCell

These two functions compute the position of each triangle in the multigrid as a function of its size and position in 3D space.

For example, using the longest edge of the triangle as a measure of its size, we will assign it to a cell that may contain a surface of area $A \cong \frac{l^2}{\sqrt{2}} \cdot N$, where N is the number of triangles we wish to have in each cell. If we assume that a cell with side 2^h will contain a surface of area approximatively $2^h \cdot 2^h$, we have:

$$A \cong \frac{l^2}{\sqrt{2}} \cdot N \cong 2^{2h}$$

and hence we will compute (i, j, k, h) as:

$$\begin{array}{ll} \textbf{\textit{scale}} & \boldsymbol{h} = \lfloor \log_2 (l^2\ N) - \frac{1}{2} \rfloor - 1 \\ \textbf{\textit{cell pos}} & \boldsymbol{i} = t_{p_x}/2^h,\ \boldsymbol{j} = t_{p_y}/2^h,\ \boldsymbol{k} = t_{p_z}/2^h \end{array}$$

where t_p is a representative point of triangle, for example its barycenter. Clearly the use of this estimation does not guarantee that in the worst case no cell will have more than N triangles. However, we do not need strict bounds on the number of triangles per cell, but only that they are not so many that they cannot be loaded in-core. If the bounds are exceedingly violated, we can modify the triangle-cell assignment by redistributing triangles in too crowded cells to a lower scale (like we normally to with top-down construction hierarchies such as the oct-tree).

AddToGridCell

When we assign a triangle to a cell we must know if its vertices are already in the multigrid and where. To this purpose we introduce the concept of *global index* as a couple (c, id) addressing the id^{th} element in an array of elements in the cell c (vertices, triangles and copies can all be addressed by a global index). We use a temporary array *GAddr* of sets of global indexes for keeping track, for each vertex being added to the multigrid, in which cell it has been put. The algorithm is straightforward (please refer to Figure 4): if the vertex v has been already added to the cell c (line 5) simply set the internal reference to its position *id* (line 6), otherwise add the vertex to the cell c and add the global index to the vertex in c to $GAddr[v]$ (line 8). Thus if the set $GAddr[v]$ contains more than one global index it means that the vertex v has been assigned to more than one cells, so it is a border vertex.

Note that to add a mesh to OCME we do not need to load it entirely in main memory but only to store a global index for each of its vertices and to run over its faces. However, if the vertices themselves are too many to allow the storage of their global indexes in main memory, the array *GAddr* is mapped to the disk. When the mesh has been inserted, *GAddr* is deleted.

CreateCopies

After adding all the triangles, $GAddr[v]$ contains all the global indices of the cells storing a copy of v. If the size of $GAddr[v]$ is greater than 1 (line 12) we know that v a border vertex, thus we increment the global counter of border vertices *nB* and add a *copy* to all the cells in $GAddr[v]$.

B. Performing local Editing actions and Committing results

An *Edit* operation consists of building an in-core mesh with the content of a region of interest (ROI), which in OCME corresponds to a set of cells $ROI = \{c_0, \ldots, c_n\}$.

The algorithm consists of the following steps:

1) Load the selected cells and the corresponding dependence set $D(ROI)$;
2) Build an in-core mesh and collapse all the vertices v_i with the same $B(v_i)$;
3) Store an attribute in each vertex and face of the in-core mesh in order to encode their global index in the OCME multigrid and mark the elements in $D(ROI) \setminus ROI$ as unmodifiable;
4) Store in a temporary data structure the global index of all of the vertices and triangles fetched for editing.

Step 3 is necessary to remember who is who when the mesh will be committed back, and also to prevent the elements in the dependence set to be modified. Note that this step requires that the data structure used for encoding the in-core mesh is able to support user-defined attributes. This is possible for all of the most well known libraries for geometry processing such as CGAL, OpenMesh and VCGLib.

Step 4 is necessary to compute the set of elements that have been removed during the in-core processing (by difference to the ones that are committed back).

Commit

The algorithm for committing is essentially the same as the algorithm to add a mesh. In principle, we could just delete from the OCME the cells loaded for editing and then re-add the mesh being committed.

However, unless the in-core editing involved some radical changes (like erasing all the elements or moving them to a complete different position), it is usually much more efficient

to update the values for elements attributes and to handle the case where the changes cause a modification to the assignment of face and vertices to the cells.

IV. Rendering an OCME dataset

The most efficient massive-meshes rendering engines maximize the throughput by organizing the data (triangles or vertices) in chunks, similar to what we do in OCME. Unfortunately, the construction of these data structures takes far too long for our purposes; moreover, these representation schemes are not naturally extendable to include interactive modification of the data.

A. A Simple Multiresolution Scheme.

A multiresolution scheme consists of two parts: a hierarchical data structure where each node of the hierarchy covers a portion of the space and a way to build an impostor representation for the data contained in the region covered by said node.
As for the hierarchy, a natural choice is the oct-tree with nodes corresponding to grid cells. The parent/children relations are implicitly defined by the multigrid data structure, i.e.:

$$\begin{array}{ll} Parent(i,j,k,h) = & (i \gg 1, j \gg 1, k \gg 1, h+1) \\ Chidren(i,j,k,h) = & \{(i \ll 1+di, j \ll 1+dj, k \ll 1+dk, h-1) \\ & \mid di,dj,dk \in [0..1]\} \end{array}$$

where $\gg$ and $\ll$ are bit-shift operations.
The impostor representation for a cell is obtained by partitioning the cell in k^3 sub-cells and computing a *geometric proxy* for each sub-cell. As geometric proxy we chose a point placed in the center or each sub-cell with a normal vector encoded with 16 bits and a 8 bits per channel RGB color. The proxies are computed during the insertion of the mesh, by collecting one sample per triangle and summing up its contribution to the current normal and color of the proxy, without requiring to reload geometry previously stored in the cell. The choice for k is related to the average number of triangles (and therefore the memory required) per cell. As a rule of thumb we use $k = \lfloor \sqrt{\frac{N}{26}} \rfloor$ which turned out to be a reasonable choice with the current hardware. Thus for $N = 5000$ we have $k = 8$ which means we use one proxy every 64 triangles.

Building the Impostor Representation.

We build a forest of oct-trees in a bottom-up fashion, starting by the lowest non empty level of the multigrid and proceeding upwards level by level. The listing in Figure 5 sketches the algorithm. The condition at line 5 says when to stop to create upper impostor on the basis of the *occupancy* of a cell. The occupancy is an estimation of the percentage of space of the cell occupied by data. If its value is under *thr_occupancy*, $Parent(c)$ is not inserted in the next level and it becomes one of the root nodes of the oct-tree forest.
The occupancy of the cell $Parent(c)$ is found as the maximum of two quantities: the average occupancy of its children, i.e. the occupancy information propagated from the lower levels of the hierarchy, and the *data occupancy*, i.e. the percentage of occupied space given be the data assigned to $Parent(c)$. The

```
1 Algorithm BuildImpostorHierarchy(Cells)
2  for(int lev = lowest_non_empty(Cells); lev < max_level; ++lev)
3   for c in non_empty_cells[lev]
4     Comp_Impostor_UpdateOccupancy(Parent(c),Siblings(c))
5     if(Occupancy(Parent(c)) > thr_occupancy)
6       non_empty_cells[lev+1] += Parent(c)
7       else
8       root_nodes += Parent(c)
```

Fig. 5. Algorithm for computing the impostor hierarchy.

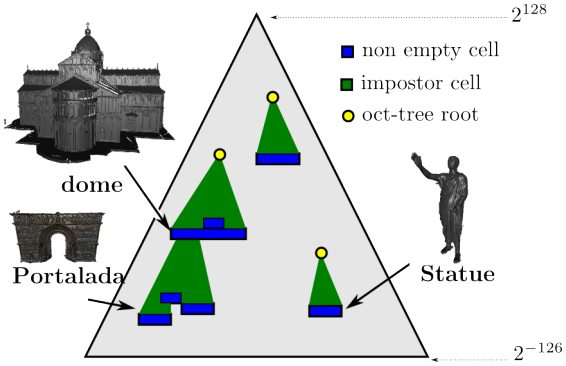

Fig. 6. Scheme of the forest of oct-trees in a multi-scale dataset.

updating of occupancy and the computation of the impostor is carried out by function at line 4.

Figure 6 shows a scheme of the forest of oct-trees for a mixed dataset. Note that the hierarchies of impostors are generated only from the impostors of the non empty cells (rendered in blue) and no loading of geometry data is involved. When a commit operation is executed, the impostors for all the cells involved in the commit are rebuilt, and the algorithm of Figure 5 is applied to them to update the correspondent portion of hierarchy.

Note that a cell may easily contain **both** geometric data **and** impostor. For example part of the cells containing the data of the Dome (please refer to Figure 6) also contain the upper part of the hierarchy of impostors for the Portada.

Rendering pass: The rendering pass consists of visiting the hierarchies top down starting from the root nodes and stopping when an error criterion is met, as in all multiresolution approaches. Since the impostors are built from a regular partition of a cell in sub-cells, we may consider the geometric approximation error, intended as the maximum distance between a proxy and the real data, bounded by the size of the sub-cell and therefore proportional to the cell size, i.e. to the cell level in the multigrid. Thus, if the projection of the cell is more than a user-defined threshold (in our setting we found 50 pixels is a reasonable choice) and the cell is not a leaf, the geometric data possibly contained in the cell are rendered and its children are visited. Wrapping up, at the end of the rendering pass we will see: the impostor representation of cells which on-screen error was under the threshold and the geometry data encountered in the visit of the oct-tree. Since our impostors are a collection of points with normals, and we know by construction the average inter-point distance, we can use the Algebraic Point Set Surfaces [5] to obtain a more graceful surface (see Figure 7).

Fig. 7. Left: the multi-resolution point cloud; Right: Algebraic Point Set Surface of the same point cloud.

B. Defining the Region of Interest

The OCME data structure is four dimensional, so the ROI must also be four dimensional, i.e. we need a way to specify the cells of the multigrid we are interested in starting from a three dimensional region specified by the user. A simplistic choice would be just to specify a 3D ROI and consider all the cells at any level whose 3D projection intersects the 3D ROI. Unfortunately it would not work: Figure 8 (top) shows a view of a 177 Mtr model of the Portada de Santa María de Ripoll, obtained by using a Minolta 910 Vi Laser scanner with a sampling resolution of 0.3 mm, which is arranged side by side with a model of the Dome of Pisa, obtained with a time-of-flight Leyca scanner at 1 point per centimeter resolution. Suppose that the red square defines the user-defined ROI: if we just tried to load the content of the cells falling in the selection we would end up trying to load the whole $177Mtr$ Portada model in main memory. Fortunately this problem is implicitly solved by our multi-resolution scheme: if a cell is rendered as impostor it means that it is the root of a subtree which leaves contains too many triangles to be rendered all together (and hence loaded for editing). In this example, the Dome is all rendered with real geometric data while the Portada is only show with impostors. Therefore when the user selects a region we only consider cells that are being rendered with geometric data and ignore those rendered as impostors (see bottom of Figure 8).

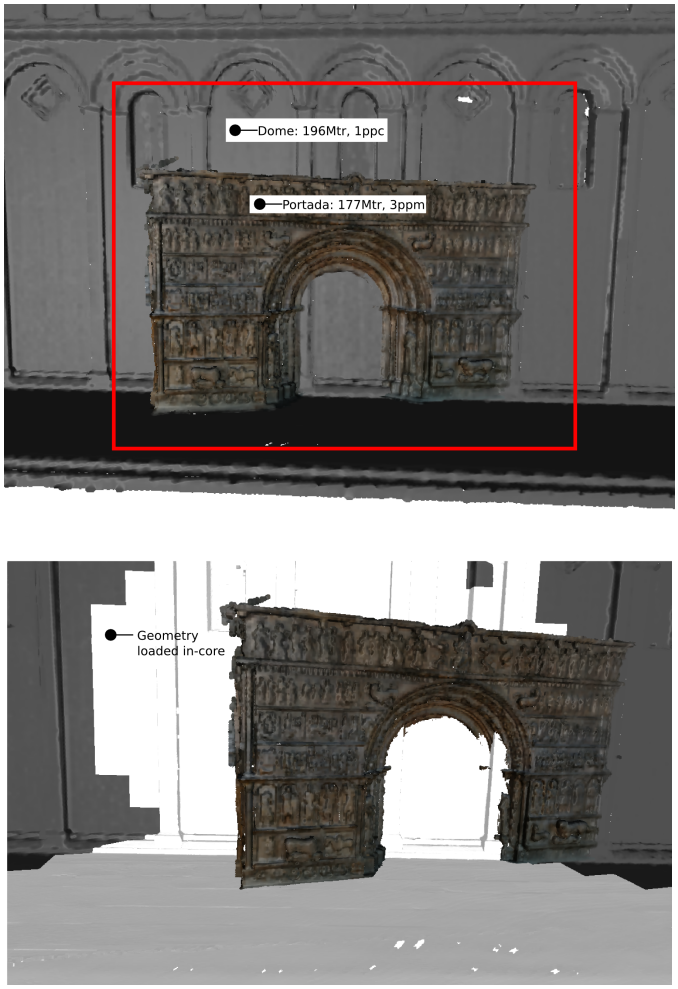

Fig. 8. Two overlapping models with different density. The size of the selection is used to determine the scale of the data that is intended to be loaded.

V. Compression and Storage

An OCME dataset is ultimately a collection of meshes, thus we can apply any existent mesh compression method to each single cell separately. However, being that we allow to add any type of attributes to vertices, the compression of the mesh is not the most critical aspect. Here we only describe the peculiar advantages derived by using the OCME data structure.

Compressing Vertex Positions

If a vertex is stored in the cell (i,j,k,h), it follows that its x coordinates will be in the interval $[i\cdot 2^h,(i+1)\cdot 2^h]$ (the same holds replacing i with j or k). Considering the IEEE floating point representation, this means that all the vertices in the same cell will have the same first $\log_2 i$ bits of the mantissa and that the exponent cannot be bigger of $2^{log_2 i+h}$, therefore there are $log_2 i+(8-\log_2 i+h)$ redundant bits. Furthermore, since we know that the size of each triangle contained in a cell at level h, we can decide to drop some bits of the mantissa with a strict bound on the precision lost.

Compressing Connectivity Since OCME tends to have grid cells with a limited number of triangles and vertices, we can encode the triangle-vertices adjacency with $\log NV$ bits, where NV is the number of vertices in the cell.

Storage All the per-triangle and per-vertex attributes are stored in out-of-core vectors. OCME implements out-of-core vectors as a sequence of fixed-size chunks of data. When a position of a vector needs to be accessed, the corresponding chunk is loaded from disk to main memory. The set of chunks loaded in memory is handled with a simple LRU cache policy. With this organization to access the n^{th} position requires a division (to find out the chunk) and one deferentiation.

Since the OCME vectors are a sequence of $(Id,Data)$ where Id identifies the vector and the chunk and $Data$ is a constant size amount of binary information, they can be naturally stored in databases such as Oracle Berkeley DB or Kyoto Cabinet, which are open source solutions for data storage management supporting transactions, so that an accidental crash can usually be recovered, a very important issue when data being handled is massive and critical.

Note that the vector for a single cell (or for a few cells) could be entirely loaded in memory without the need to organize them in chunks, because of the average bounded amount of elements associated with each cell. However, when adding a mesh to OCME we must consider that the triangles can potentially be distributed among a large number of cells, so that we would have to load/unload entire vectors only to modify a small part of them. As a limit case, consider adding one triangle to each one of the cells already created: it would mean to load and save the entire dataset.

VI. Results and Discussion

We ran a number of tests on a Intel Core Duo CPU 2.33 Ghz 3GB and a hard disk serial ATA 7200 rpm equipped with Windows 7 32 bit OS. The first tests are related to the insertion of a mesh into the database. From Table I we can see that the OCME construction is fast. Even for quite large amount of data, requiring access to a high number of cells, we can build data structure at the pace of 150K triangles per second, and almost half of this time is spent for loading the data from a binary PLY file (although admittedly this percentage would be different if loading from a more efficient data format). Table II reports the time and disk size if redundancy of floating point representation is eliminated as explained in section V. We created 3 versions of the Dome, scaling it size to the unitary box and placing its center to three different positions in space. As expected, the farther away the model is from the origin (i.e. the more common bits in the mantissa) the more disk space is saved. However, the operations to eliminate the redundant bits (maskings and bit shifts) from each floating point representation take time and slow down the insertion, which must be considered when choosing if using or not suppression of redundant bits.

Figure 9 shows the distribution of the number of triangles (top) and size of the dependent set over the cells (bottom) on the Portada dataset for a user defined value $N = 5000$. It may be noted as in the great majority of cases the triangles assigned to the cells are between 4000 and 6000, and no cell has more that 11000 triangles. Note that while we have an average upper bound on the maximum number of triangles per cell, we may have many cells with a small number of triangles, depending on how the meshes are positioned in the grid. Typically we may have some peaks like those in the left part of the graph corresponding to cells which are only partially overlapped the border of the Portada dataset. As expected, the size of the dependent set, which is almost constantly around 9, shows that a cell is generally dependent on its immediate neighbors at the same level.

For evaluating the edit/commit time we run a series of random edit-commit operations by operating a laplacian smoothing or an edge collapse decimation. For each test, we select a cell with uniform probability among the those containing data and load it (and its dependent set) for editing. Figure 10 (top) shows that OCME retrieves from secondary memory roughly $100K$ triangles per second and commits back previously edited meshes at a pace of $30K$ to $50K$ triangles per second. We performed the test for simplification and smoothing to show

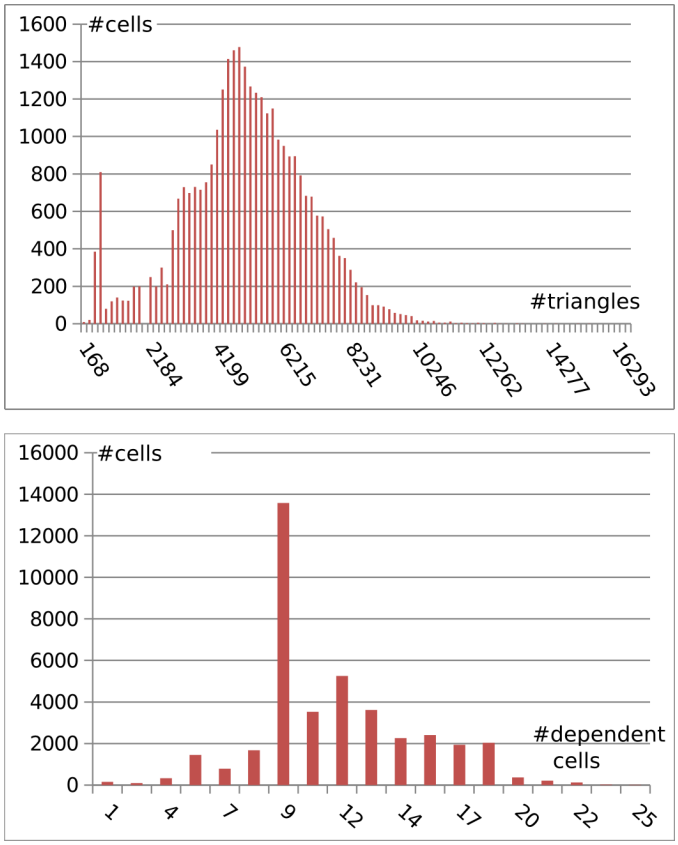

Fig. 9. Distribution of the number of triangles (top) and distribution of the number of dependent cells for $N = 5000$

that in the first case, although the decimation process brings to fewer triangles to commit, the time required for committing is longer. This is due to the fact that the removal of collapsed faces causes fragmentation in the vectors storing the data and that many of the faces that are not deleted by decimation are likely to change level of the multigrid because they have become bigger.

Comparison with existing techniques

OCME is neither a compression algorithm nor a rendering technique, thus it would compare poorly against ad hoc solutions to these problems. The Random Accessible Triangle Meshes [9], for example, shows a construction time of $380K$ triangle per second, which is more than two times faster than OCME and allows random access to the mesh, but on read-only mode. The Adaptive Tetrapuzzle [4] renders the full detail of large meshes at above 50 fps but again the original data cannot be modified and the processing time is an order of magnitude bigger that OCME. For example, the Adaptive TetraPuzzle takes 408 minutes to process a mesh composed by 56 million triangles, while the conversion to OCME data structure takes only 20 minutes for a model of 177 million triangles. The solution most similar to OCME, at least in terms of functionalities, is the system by Wand et al. [10], since it allows the interactive editing of large datasets, although only for point clouds. In their work they report an insertion time of $74.4K$ points per second on a very similar machine architecture (we only used a single core in our implementation and tests)

				Size disk (GB)		OCME data		time (sec)			
	tri	vert	#files	input	output	#cells	#impost.	mesh load	disk R	disk W	Total
Aulo Metello	93.7	45.6	298	2.35	2.67	8.573	582.561	256.88	116.4	178.7	960
Dome	196.0	98.2	372	3.748	3.634	22589	1,4M	447.9	53.9	170.1	1141.8
Portada	177.4	89.7	405	3.742	3.65	55.556	3,8M	470.6	55.1	186.5	1180.4
Composition	430.1	217.4	784	8.674	9.60	27208	35970	1,173	389.7	505.8	3205.2

TABLE I
RESULTS FOR THE INSERTION OF MESHES IN THE OCME DATA STRUCTURE FOR N=5000.

pos (i,j,k)	(0,0,0)	(1024,1024,1024)	(1048576,1048576,1048576)
size (MB)	3634	3340	3028
ins. time (s)	1141.8	1973	1803

TABLE II
PERFORMANCES FOR THE INSERTION OF THE DOME IF THE REDUNDANT BITS OF FLOATING POINT VERTEX POSITIONS ARE ELIMINATED.

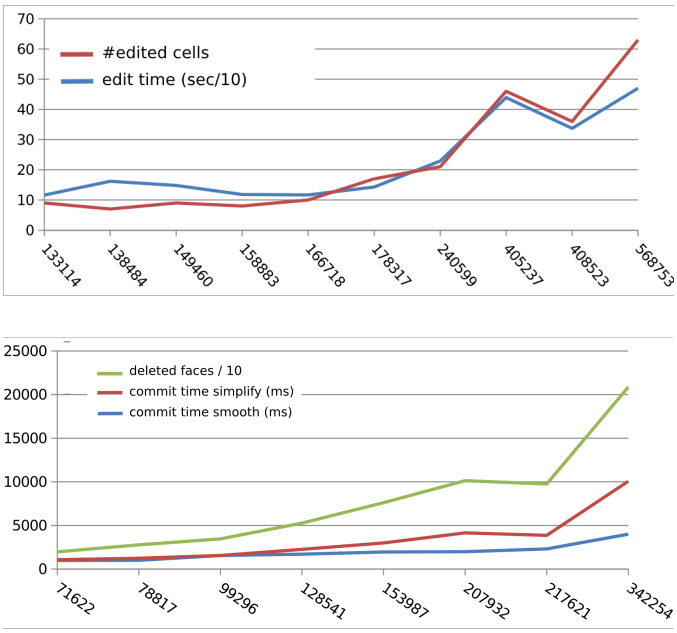

Fig. 10. Time for taking a portion of dataset in-core for editing (top); time for committing a mesh from in-core after a decimation or a smoothing

for a dataset of 75.7 million points, while we are able to insert 134K triangles per second for a dataset of 522M triangles.

A. Limitations

OCME is designed for real world cases with special focus on scanned models and exploits the fact that the density of the surface does not chance much locally, and that it is proportional to the surface per volume. If we violate these assumptions, for example because we insert multiple copies of the same triangle mesh, we will experiment a performance loss until the system will not be able to build a mesh because some cell will contain too much data to be loaded in-core. We can avoid this degeneration by checking the number of triangles per cell and redistributing the triangles to lower levels grid for too crowded cells in a hierarchical fashion to prevent OCME from stop working, but it is clear that we would not do any better than a standard octree partition of data.

B. Extensions

Although we illustrated OCME for the special case of triangle meshes, it must be said that it works with evey kind of geometrical data that carries a local *measure* of size and a position. For example we can store generic polygons, segments, boxes and so on. Among others, point clouds are of particular interest because they are the actual data coming from the scanning device. We support the addition of point clouds by estimating the radius of each point as the average distance of its closest 10 neighbors and using it as size of the point. Clearly point clouds never involve border vertices since there is no explicit connectivity, thus there is no data duplication. In this sense we cannot talk of an extension to the case of point clouds because it is actually an under-use of the framework.

VII. CONCLUSIONS AND FUTURE WORK

We presented OCME, a simple framework for interactive editing of large spurious datasets. In contrast to previous work, OCME is intended as a solution for supporting editing over large 3D datasets as those produced by current 3D acquisition technologies. OCME does not requires a once-for-all preprocessing step to build its data structure and supports local definitions of generic non-geometric attributes.
There are several parts of OCME that can be improved without changing the core of the framework. The most straightforward optimization consists of introducing mesh compression solutions for encoding the sub-meshes contained in the single cells. A more intriguing matter is the possibility of remote and collaborative work on the same dataset, which could dramatically speed up the process of mesh repairing that is necessary in every acquisition campaign. From a system development point of view, we plan to extend the OCME interface toward CGAL and OpenMesh data structures, while at the moment only VCGLib meshes are supported. An implementation of OCME is freely available as a MeshLab plugin [3].

VIII. ACKNOWLEDGEMENTS

We thank our colleagues Marco Di Benedetto and Federico Ponchio for the insightful discussions on open source

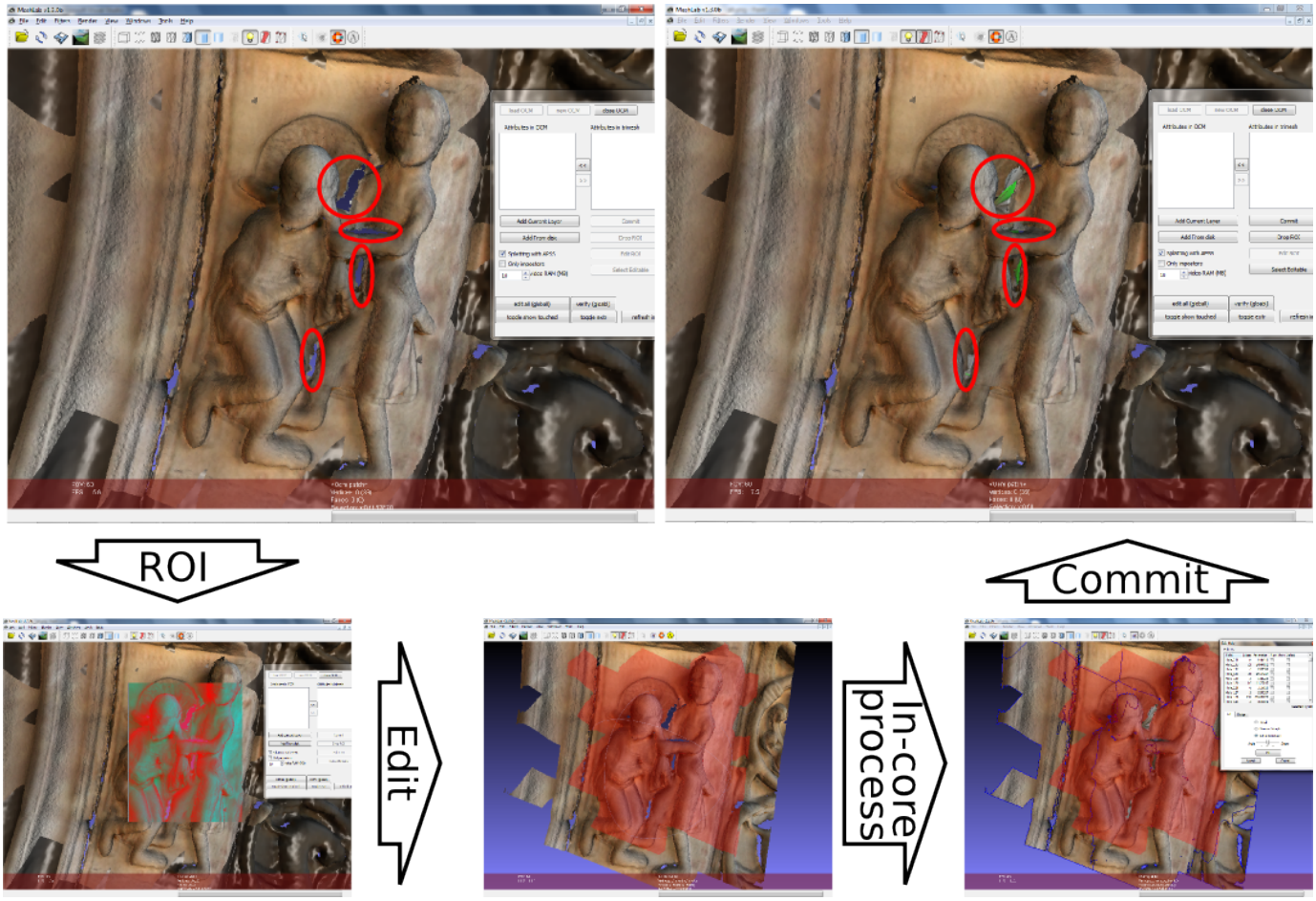

Fig. 11. Example of edit / in-core processing / commit to fill four holes in the mesh.

databases and caching techniques. The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 231809 (IST IP "3DCOFORM").

AUTHORS

Fabio Ganovelli is a research scientist at the Istituto di Scienza e Tecnologie dell'Informazione (ISTI) of the National Research Council (CNR) in Pisa, Italy. He received an advanced degree in Computer Science (Laurea) in 1995 and a PhD in Computer Science from the University of Pisa in 2001. His research interests include modelling of deformable objects, geometry processing and scientific visualization, with special focus on rendering of large datasets.

Roberto Scopigno is a Research Director with CNR-ISTI and leads the Visual Computing Lab. He graduated in Computer Science at the University of Pisa in 1984. He is engaged in research projects concerned with 3D scanning, surface reconstruction, multiresolution data modeling and rendering, scientific visualization and applications to cultural heritage. He published more than hundred fifty papers in international refereed journals/conferences and gave invited lectures or courses on visualization and graphics at several international conferences. Roberto has been responsible person for CNR-ISTI in several EU projects. He was Co-Chair of international conferences (Eurographics '99, Rendering Symposium 2002, WSCG 2004, Geometry Processing Symp. 2004 and 2006, VAST 2005, Eurographics 2008) and served in the programme committees of several events (Eurographics, ACM Siggraph, IEEE Visualization, EG SGP, etc.). He is Chair of the Eurographics Association (2009-2010), Co-Editor in Chief of the journal Computer Graphics Forum and member of the Editorial Board of the ACM J. on Computing and Cultural Heritage.

REFERENCES

[1] T. Boubekeur and C. Schlick, "Interactive out-of-core texturing with point-sampled textures," in *Symposium on Point-Based Graphics*, Eurographics Association, 2006, pp.67–73.

[2] P. Cignoni and R. Scopigno, "Sampled 3d models for ch applications: an enabling medium or a technological exercise?" *ACM Journ. on Computers and Cultural heritag*, vol. 1, no. 1, pp. 2:1–2:23, 2008.

[3] P. Cignoni, M. Callieri, M. Corsini, M. Dellepiane, F. Ganovelli, and G. Ranzuglia, "Meshlab: an open-source mesh processing tool," in *Sixth Eurographics Italian Chapter Conference*, 2008, pp. 129–136.

[4] P. Cignoni, F. Ganovelli, E. Gobbetti, F. Marton, F. Ponchio, and R. Scopigno, "Adaptive tetrapuzzles: Efficient out-of-core construction and visualization of gigantic multiresolution polygonal models," *ACM Trans. on Graphics (SIGGRAPH 2004)*, vol. 23, pp. 796–803, 2004.

[5] G. Guennebaud and M. Gross, "Algebraic point set surfaces," *ACM Trans. Graph.*, vol. 26, July 2007.

[6] J. Ho, K. Lee, and D. Kriegman, “Compressing large polygonal models,” in *Presented at IEEE Visualization*, vol. 2001, 2001, pp. 357–362.

[7] J. Peng, C. Kim, C. Jay Kuo,“Technologies for 3D mesh compression: A Survey,” *Journal of Visual Communication and Image Representation*, vol. 16, no. 6, pp. 688-733, 2005.

[8] S.E. Yoon, E. Gobbetti, D.J. Kasik, and D. Manocha “Real-Time Massive Model Rendering” *Synthesis Lectures on Computer Graphics and Animation*, Morgan & Claypool Publishers, 2008

[9] S.-E. Yoon and P. Lindstrom, “Random-accessible compressed triangle meshes,” *IEEE Trans. Vis. Comput. Graph*, vol. 13, no. 6, pp. 1536–1543, 2007.

[10] M. Wand, A. Berner, M. Bokeloh, A. Fleck, M. Hoffmann,P. Jenke, B. Maier, D. Staneker, and A. Schilling “Interactive editing of large point clouds,” *Proceedings of Symposium on Point-Based Graphics (PBG 07)*, 2007 pp.37–46.