CHuMI Viewer: Compressive Huge Mesh Interactive Viewer

Clément Jamin[*,a,b], Pierre-Marie Gandoin[a,c], Samir Akkouche[a,b]

[a] *Université de Lyon, CNRS*
[b] *Université Lyon 1, LIRIS, UMR5205, F-69622, France*
[c] *Université Lyon 2, LIRIS, UMR5205, F-69676, France*

Abstract

The preprocessing of large meshes to provide and optimize interactive visualization implies a complete reorganization that often introduces significant data growth. This is detrimental to storage and network transmission, but in the near future could also affect the efficiency of the visualization process itself, because of the increasing gap between computing times and external access times. In this article, we attempt to reconcile lossless compression and visualization by proposing a data structure that radically reduces the size of the object while supporting a fast interactive navigation based on a viewing distance criterion. In addition to this double capability, this method works out-of-core and can handle meshes containing several hundred million vertices. Furthermore, it presents the advantage of dealing with any n-dimensional simplicial complex, including triangle soups or volumetric meshes, and provides a significant rate-distortion improvement. The performance attained is near state-of-the-art in terms of the compression ratio as well as the visualization frame rates, offering a unique combination that can be useful in numerous applications.

Key words: Lossless compression, Interactive visualization, Large meshes, Out-of-core

1. Introduction

Mesh compression and mesh visualization are two fields of computer graphics that are particularly active today, whose constraints and goals are usually incompatible, even contradictory. The reduction of redundancy often goes through signal complexification, because of prediction mechanisms whose efficiency is directly related to the depth of the analysis. This additional logical layer inevitably slows down the data access and creates conflicts with the speed requirements of real-time visualization. Conversely, for efficient navigation through a mesh integrating the user's interactions dynamically, the signal must be carefully prepared and hierarchically structured. This generally introduces a high level of redundancy and sometimes comes with data loss, if the original vertices and polyhedra are approximated by simpler geometrical primitives. In addition to the on-disk storage and network transmission issues, the data growth implied by this kind of preprocessing could become detrimental to the visualization itself since the gap between the processing times and the access times from external memory is increasing.

In this article, we attempt to reconcile compression and interactive visualization by proposing a method that combines good performance in terms of both the compression ratio and the visualization frame rates. The interaction requirements of a viewer and a virtual reality system are quite similar in terms of the computational challenge but the two applications use different criteria to decide how a model is viewed and updated. Our goal here is to take advantage of excellent view independent compression and introduce distance-dependent updates of the model during the visualization. As a starting point, the in-core progressive and lossless compression algorithm introduced by Gandoin and Devillers [1] has been chosen. On top of its competitive compression ratios, out-of-core and LOD capabilities have been added to handle meshes with no size limitations and allow local refinements on demand by loading the necessary and sufficient data for an accurate real-time rendering of any subset of the mesh. To meet these goals, the basic idea consists in subdividing the original object into a tree of independent meshes. This partitioning is undertaken by introducing a primary hierarchical structure (an *n*SP-tree) in which the original data structures (a kd-tree coupled to a simplicial complex) are embedded, in a way that optimizes the bit distribution between geometry and connectivity and removes the undesirable block effects of kd-tree approaches.

After a study of related works (Sec. 2) focusing on the method chosen as the starting point (Sec. 3), our contribution is introduced by an example (Sec. 4), then detailed in two complementary sections. First, the algorithms and data structures of the out-of-core construction and compression are introduced (Sec. 5 and 6), then the visualization point of view is adopted to complete the description (Sec. 7). Finally, experimental results are presented and the method is compared to prior art (Sec. 8).

2. Previous Works

2.1. Compression

Mesh compression is a domain situated between computational geometry and standard data compression. It consists in

*Corresponding author. Tel.: +33472448064.
Email addresses: `clement.jamin@liris.cnrs.fr` (Clément Jamin), `pierre-marie.gandoin@liris.cnrs.fr` (Pierre-Marie Gandoin), `samir.akkouche@liris.cnrs.fr` (Samir Akkouche)

efficiently coupling geometry encoding (the vertex positions) and connectivity encoding (the relations between vertices), and often addresses manifold triangular surface models. We distinguish single-rate algorithms, which require full decoding to visualize the object, and multiresolution methods that allow one to see the model progressively refined while decoding. Although historically geometric compression began with single-rate algorithms, we have chosen not to detail these methods here. The reader can refer to surveys [2, 3] for more information. For comparison purposes, since progressive and out-of-core compression methods are very rare, Sec. 8 will refer to Isenburg and Gumhold's well-known single-rate out-of-core algorithm [4]. Similarly, lossy compression, whose general principle consists in a frequential analysis of the mesh, is excluded from this study.

Progressive compression is based on the notion of refinement. At any time of the decoding process, it is possible to obtain a global approximation of the original model, which can be useful for large meshes or for network transmission. This research field has been very productive for approximately 10 years, and rather than being exhaustive, here the choice has been to adopt a historical point of view. Early techniques of progressive visualization, based on mesh simplification, were not compression-oriented and often induced a significant increase in the file size, due to the additional storing cost of a hierarchical structure [5, 6]. Afterward, several single-resolution methods were extended to progressive compression. For example, Taubin *et al.* [7] proposed a progressive encoding based on Taubin and Rossignac's algorithm [8]. Cohen-Or *et al.* [9] used techniques of sequential simplification by vertex suppression for connectivity, combined with position prediction for geometry. Alliez and Desbrun [10] proposed an algorithm based on progressive removal of independent vertices, with a retriangulation step under the constraint of maintaining the vertex degrees around 6. Contrary to the majority of compression methods, Gandoin and Devillers [1] gave the priority to geometry encoding. Their algorithm, detailed in Sec. 3, gives competitive compression rates and can handle simplicial complexes in any dimension, from manifold regular meshes to triangle soups. Peng and Kuo [11] took this paper as a basis to improve the compression ratios using efficient prediction schemes (sizes decrease by roughly 15%), still limiting the scope to triangular models.

2.2. *Large meshes processing*

Working with huge datasets implies to develop out-of-core solutions for managing, compressing or editing meshes. Lindstrom and Silva [12] proposed a simplication algorithm based on [13] whose memory complexity is neither dependent on the size of the input nor on the size of the output. Cignoni *et al.* [14] introduced a data structure called Octree-based External Memory Mesh (OEMM), which enable external memory management and simplification of very large triangle meshes, by loading only selected sections in main memory. Isenburg *et al.* [15] developed a streaming file format for polygon meshes designed to work with large data sets. Isenburg *et al.* [16] proposed a single-rate streaming compression scheme that encodes and decodes arbitrary large meshes using only minimal memory resources. Cai *et al.* [17] introduced the first progressive compression method adapted to very large meshes, offering a way to add out-of-core capability to most of the existing progressive algorithms based on octrees. The next section pays particular attention to out-of-core visualization techniques.

2.3. *Visualization*

Fast and interactive visualization of large meshes is a very active research field. Generally, a tree or a graph is built to handle a hierarchical structure which makes it possible to design out-of-core algorithms: at any moment, only necessary and sufficient data are loaded into memory to render the mesh. Level of detail, view frustum and occlusion culling are widely used to adapt displayed data to the viewport. Rusinkiewicz and Levoy [18] introduced QSplat, the first out-of-core point-based rendering system, where the points are spread in a hierarchical structure of bounding spheres. This structure can easily handle levels of detail and is well-suited to visibility and occlusion tests. Several million points per second can thus be displayed using adaptive rendering. El-Sana and Chiang [19] proposed a technique to segment triangle meshes into view-dependence trees, allowing external memory simplification of models containing a few millions polygons, while preserving an optimal edge collapse order to enhance the image quality. Later, Lindstrom [20] developed a method for interactive visualization of huge meshes. An octree is used to dispatch the triangles into clusters and to build a multiresolution hierarchy. A quadric error metric is used to choose the representative point positions for each level of detail, and the refinement is guided by visibility and screen space error. Yoon *et al.* [21] proposed a similar algorithm with a bounded memory footprint: a cluster hierarchy is built, each cluster containing a progressive submesh to smooth the transition between the levels of detail. Cignoni *et al.* [22] used a hierarchy based on the recursive subdivision of tetrahedra in order to partition space and guarantee varying borders between clusters during refinement. The initial construction phase is parallelizable, and GPU is efficiently used to improve frame rates. Gobbetti and Marton [23] introduced the *far voxels*, capable of rendering regular meshes as well as triangles soups. The principle is to transform volumetric subparts of the model into compact direction-dependent approximations of their appearance when viewed from a distance. A BSP tree is built, and nodes are discretized into cubic voxels containing these approximations. Again, the GPU is widely used to lighten the CPU load and improve performance. Cignoni *et al.* [24] proposed a general formalized framework that encompasses all these visualization methods based on batched rendering. Recently, Hu *et al.* [25] introduced the first highly parallel scheme for view-dependent visualization that is implemented entirely on programmable GPU. Although it uses a static data structure requiring 57% more memory than an index triangle list, it allows real-time exploration of huge meshes.

2.4. *Combined Compression and Visualization*

Progressive compression methods are now mature (the rates obtained are close to theoretical bounds) and interactive visualization of huge meshes has been possible for several years.

However, even if the combination of compression and visualization is often mentioned as a perspective, very few papers deal with this problem, and the files created by visualization algorithms are often much larger than the original ones. In fact, compression favors a small file size to the detriment of fast data access, whereas visualization methods focus on rendering speed: the two goals oppose and compete with each other. It must be noted that there are a few counter-examples such as [26] where there is a reduction of redundancy without any runtime penalty, but it remains unusual. Among the few works that introduce compression into visualization methods, Namane *et al.* [27] proposed a QSplat compressed version by using Huffman and differential coding for spheres (position, radius) and normals. The compression ratio is approximately 50% compared to the original QSplat files, but the scope is limited to point-based rendering. More recently, Yoon and Lindstrom [28] introduced a triangle mesh compression algorithm that supports random access to the underlying compressed mesh. However, their file format is not progressive and therefore inappropriate for global views of a model since it would require loading the full mesh.

3. The Starting Point: kd-tree Compression of Vertex Position and Hierarchical Connectivity Coding

In this section, we briefly present the method originally proposed by Gandoin and Devillers [1]. The algorithm, valid in any dimension, is based on a top-down kd-tree construction by cell subdivision. The root cell is the bounding box of the point set and the first output is the total number of points. Then, for each cell subdivision, only the number of points in the first half-cell has to be coded, the content of the second half-cell being easily deduced. As the algorithm progresses, the cell size decreases and the data transmitted provide a more accurate localization of the points. The subdivision process iterates until there is no longer a non-empty cell with a side greater than 1 spatial unit, such that every point is localized with the full mesh precision.

As soon as the points are separated, the connectivity of the model is embedded (one vertex per cell) and the splitting process is run backwards: the cells are merged bottom-up and their connectivity is updated. The connectivity changes between two successive versions of the model and is encoded by symbols inserted between the geometry codes. The vertices are merged by the two following decimating operators:

- Edge collapse, originally defined by Hoppe *et al.* [5] and widely used in surface simplification, merges two adjacent cells under certain hypotheses. The endpoints of the edge are merged, which leads to the deletion of the two adjacent triangles (only one if the edge belongs to a mesh boundary) (see Fig. 1a).

- Vertex unification, as defined by Popović and Hoppe [6], is a more general operation that merges any two cells even if they are not adjacent in the current connectivity. The result is non-manifold in general (see Fig. 1b) and the corresponding coding sequence is approximately twice as big as an edge collapse coding sequence.

The way the connectivity evolves during these decimating operations is encoded by a sequence that enables a lossless reconstruction using the reverse operators (edge expansion and vertex split).

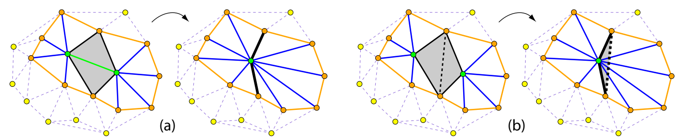

Figure 1: (a) Edge collapse, (b) Vertex unification

If this lossless compression method reaches competitive ratios and can handle arbitrary simplicial complexes, it is not appropriate for interactive navigation through large meshes. Not only does its memory footprint make it strictly impracticable for meshes over one million vertices, but, even more constraining, the intrinsic design of the data structure imposes a hierarchical coding of the neighborhood relations that prevents any local refinement: indeed, given 2 connected vertices v and w in any intermediate level of the kd-tree, it is possible to find a descendant v_i of v that is connected to a descendant w_j of w. Therefore, in terms of connectivity, the cell containing v cannot be refined without refining the cell containing w. Consequently, random access and selective loading of mesh subsets is impossible: to visualize a single vertex and its neighborhood at a given LOD, the previous LOD must be entirely decoded. In the following, algorithms and data structures are presented that are based on the same kd-tree approach but remove these limitations.

4. Overview

The compression process of *CHuMI* begins by partitioning the space, creating a hierarchical structure called an *n*SP-tree in which the simplices are dispatched. In order to avoid SP-cells containing only a few triangles, an SP-cell must contain at least N_{min} (user defined) triangles to be split. Moreover, to allow the independent decoding of any vertex with its neighborhood, the simplices whose vertices belong to several SP-cells are duplicated into each cell. We then traverse the SP-tree in postorder (a cell is processed after all its children have been processed). In each SP-cell, incident kd-cells are progressively merged as described in Sec. 3, and the corresponding coding sequence is constructed, where geometry and connectivity codes are interleaved. Merging proceeds until the minimal precision of the SP-cell is reached: in the example given in Fig. 2, this corresponds to one kd-cell merging in each dimension for a regular SP-cell, and 3 mergings in each dimension for the root SP-cell. When all the children of an SP-cell s have been treated, the submeshes they contain are merged together (the duplicated simplices collapse) to form the submesh associated with s, which is then ready to be treated. This process runs until a single vertex remains in the root SP-cell.

The refinement process runs backward. Figure 2 illustrates its main stages: steps 1 to 6 are similar to [1]: we start with a null precision and a single centered point. Then we sequentially read the geometry and connectivity codes from the compressed

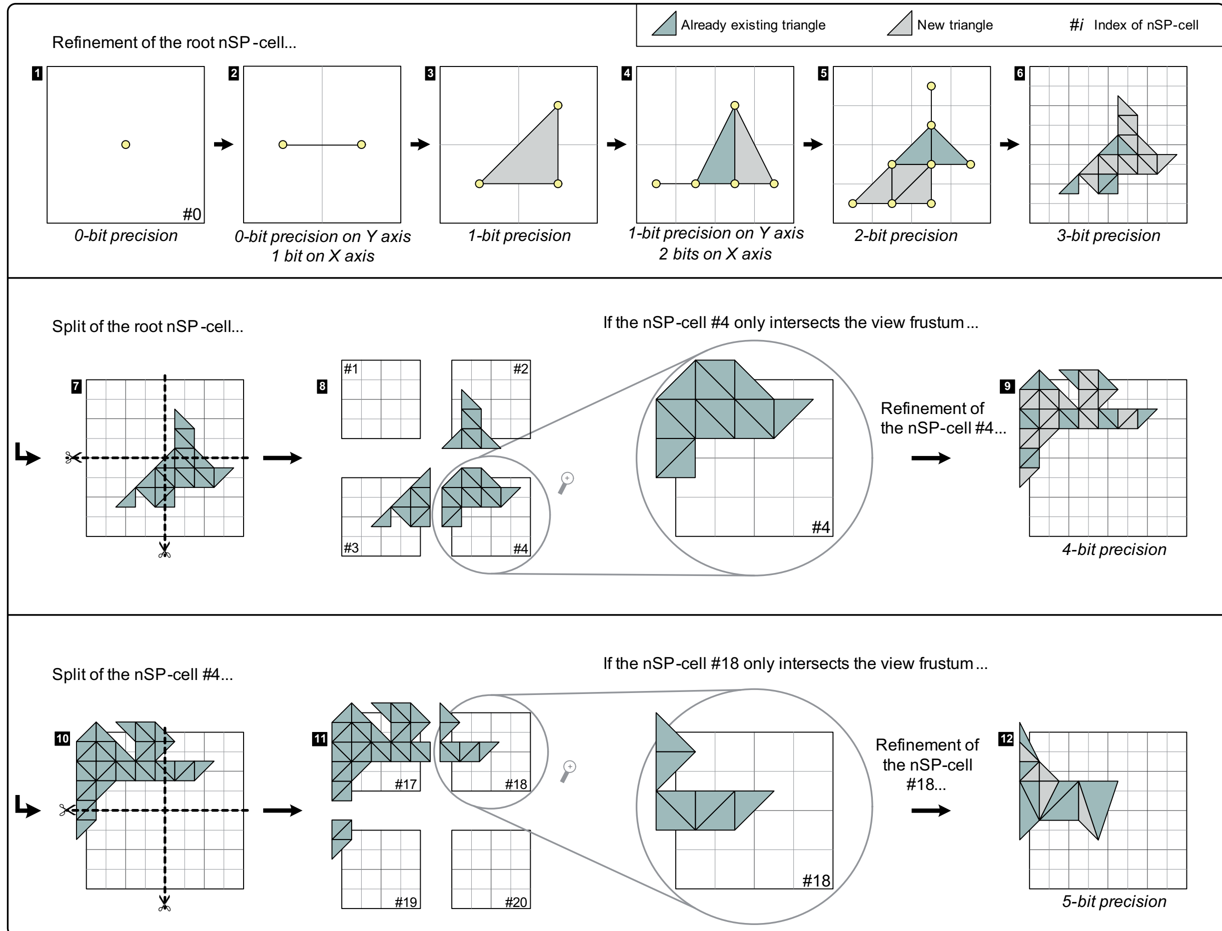

Figure 2: Overview of the method

file, which progressively refine the entire mesh. In this example, when precision reaches 3 bits (in practical cases, we generally wait until precision reaches 7 or 8 bits), the space is split into 4 subspaces, and the mesh subdivided into these subspaces. Remember that border triangles and their associated kd-cells have been duplicated during the compression stage. The 4 submeshes obtained become independent, each one corresponding to a subspace called an SP-cell whose content is directly accessible from an index pointing in the compressed file. Thus we can select one or several SP-cells and read their associated refinement codes in the compressed file in order to continue the refinement process locally, depending on the camera moves. When one of them reaches its maximum precision (4 bits in our example), it is split again. This process is repeated recursively until each SP-cell in the view frustum reaches a level of precision determined by its distance to the camera. Finally, we connect the SP-cells together, handling the possible differences in resolution to prevent border triangles from overlapping, and the content of these cells is sent to the graphic pipeline.

5. Out-of-Core Construction

Before describing the construction process itself, we begin with a few definitions:

An nSP-tree is a space-partitioning tree with n subdivisions per axis. Each node delimits a cubic subspace called the nSP-cell. Since our meshes are embedded in $3D$, an internal nSP-cell has n^3 children (wherever it is in the tree), and in our case, the nSP-cells are evenly split, *i.e.* their n^3 children are the same size.

Each nSP-cell c contains vertices whose precision at runtime can vary according to the distance from c to the camera. The boundaries of the vertex precision in c are called minimal and maximal precision of c.

A top-simplex is a simplex (vertex, edge, triangle or tetrahedron) that does not own any parent, *i.e.* that does not compose any other simplex. In a triangle-based mesh, the top-simplices are mainly triangles, but in a general simplicial complex, they can be of any type.

5.1. Quantizing point coordinates

First, the points contained in the input file are parsed. A cubic bounding box is extracted, then the points are quantized to obtain integer coordinates relative to the box, according to the maximum level of precision p (number of bits per coordinate) requested by the user. It should be noted that this quantization step does not contradict the lossless characteristic of the method since p can be chosen as high as needed to match the object's

initial precision. These points are written to a raw file (RWP) using a classical binary encoding with constant size codes, such that the ith point can be directly accessed by file seeking.

5.2. *nSP-tree construction*

An nSP-tree is built so that each leaf contains the set of the top-simplices lying in the corresponding subspace. The height of the nSP-tree depends on the mesh precision p (the number of bits per coordinate): the user can choose the maximal precision p_r of the root, then the precision p_l gained by each next level is computed from n ($p_l = log_2 n$). For example, for $p = 12$ bits, $p_r = 6$ bits and $n = 4$, the nSP-tree height is 4. Another condition is applied: an nSP-cell can be split only if the number of contained top-simplices is at least equal to a threshold N_{min}. Consequently, the minimal precision of the leaves can take any value, whereas their maximal precision is always p. The content of the leaves is stored in a raw index file (RWI) where the $d + 1$ vertex indices that compose a d-simplex are written using a raw binary format to minimize the disk footprint while maintaining a fast and simple reading. Fig. 4 presents a 2D example of such an nSP-tree.

Since an nSP-cell is only subdivided when it contains more than N_{min} triangles, the nSP-tree takes into account the local density of the mesh: sparse regions are hardly divided, whereas high density areas are heavily splitted. This simple scheme has been chosen because it combines adaptability and low encoding cost.

5.3. *RWP and RWI file management*

Even if the points and triangles are not ordered in the input file, it is likely that the different accesses to a given point of the RWP file during the nSP-tree construction stage will be close in time. Consequently, the algorithm uses a cache in the main memory, based on a hash table, to minimize the disk accesses to vertex positions. The hash keys are modulos of the point indices and guarantee a constant time access to cached points.

Since the final number of the triangles contained in an nSP-cell is not known in advance, memory cannot be allocated statically for each cell in the RWI file. Consequently, the algorithm has to use block list management: the RWI file is split into constant size blocks (containing 100 triangles for instance), and the content of an nSP-cell is stored in a chained list of such blocks. As shown in Fig. 3, each nSP-cell keeps 3 numbers in main memory: the offset of its head block in the RWI file, the offset of the current position (the location where the next triangle will be written in the RWI file), and the number of triangles contained in the current block (to know when a new block must be allocated). Each block contains the offset of its successor (or −1 for the tail block of a list) and a triangle list composed of vertex indices pointing to the RWP file. Rather than writing in the RWI file for each triangle read in the input file, each nSP-cell manages a buffer containing the last added triangles, and the disk is accessed only when the cumulative size of all the buffers exceeds some fixed value b_{max}. Then the buffers are flushed in the RWI file, from the largest to the smallest, to favor sequential writing on the disk, until their cumulative size is brought back to $b_{max}/2$. When the content of an nSP-cell c reaches N_{min} triangles, c is split and the triangles (in the buffer and in the block list of the RWI file) are spread out in the newly created nSP-cells. The blocks of c are released and added to a list of available blocks used for future allocations.

5.4. *Choosing n*

In order to obtain an integer for p_l, n must be chosen as a power of 2. This guarantees that any descendant of a kd-cell contained in an nSP-cell c remains inside c. Furthermore, it prevents the kd-tree cells from overlapping two nSP-cells.

5.5. *Simplex duplication*

The first way to determine whether a simplex belongs to an nSP-cell c, consists in checking whether at least one of its incident vertices lies inside c. This test is fast and suitable for most cases, but it is imperfect since a cell can be overlapped by a simplex without containing any of its vertices. This occurs very rarely and can be perceived visually even more infrequently, but this could be an issue for certain specific applications, in which case a more accurate simplex to axis-aligned box intersection test could be performed.

Independently of the chosen test, a simplex of the original model can simultaneously belong to several nSP-cells; consequently, some simplices and their associated kd-cells must be duplicated during the distribution in the leaves to prevent holes during visualization. Although this technique degrades slightly compression ratios, it has been chosen for efficiency purposes. Drawings 7—8 and 10—11 in Fig. 2 show the space partitioning of an nSP-cell.

6. Out-of-Core Compression

In this section, we detail the successive steps of the compression process, then discuss various aspects of the implementation that contribute to reaching good performance.

To achieve competitive compression ratios, the algorithm requires a two-pass coding that makes use of two files: a semi-compressed temporary file (SCT) and the final entropy coded file (FEC).

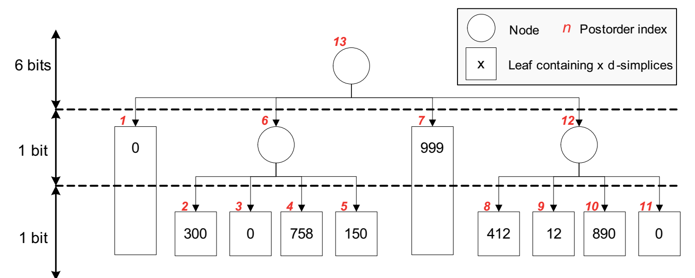

Figure 4: 2D nSP-tree example with $p = 8$ bits, $p_r = 6$ bits, $n = 2$ and $N_{min} = 1000$

	RAM	Disk
Structures	**Each SP-cell contains one:** SPCell RWI Info	**The RWI file contains a sequence of:** RWI Block
Example	**SP-cells**	**RWI file**

SPCell RWI Info:

First block file position	Current file position	Num triangles in current block
8 bytes	8 bytes	4 bytes

RWI Block:

Next block position or *-1*	List of k triangles ($k \leq 100$)
8 bytes	$3 \times 4 \times 100$ = 1200 bytes

SP-cells:

SPCell #0	0	5068	19
SPCell #1	1208	1288	6
SPCell #2	3624	7116	89

RWI file:

0		1208		2416	
2416	*100* triangles	*-1*	*6* triangles	*4832*	*100* triangles
3624		**4832**		**6040**	
6040	*100* triangles	*-1*	*19* triangles	*-1*	*89* triangles

Figure 3: RWI file structure

Writing the header of the FEC file. Regardless of the statistical data that are gathered for the final compression, some parameters characterizing the original model and the nSP-tree can be written to the FEC file at this stage. The file begins with a header that contains all general data required by the decoder: the coordinate system (origin and grid resolution, to reconstitute the exact original vertex positions), the dimension d of the mesh, the number p of bits per coordinate, the number n of subdivisions per axis for the space partitioning, as well as the root and level precision, p_r and p_l.

6.1. Treating each cell of the nSP-tree

Then the nSP-tree is traversed in postorder (in red in Fig. 4) and the following treatment is applied to each nSP-cell c.

(a) Building the kd-tree and the simplicial complex. If c is a leaf: the top-simplices belonging to c are read from the RWI file and the associated vertices from the RWP file; then a kd-tree is built that separates the vertices (top-down process) and finally the simplicial complex based on the kd-tree leaves is generated.

(b) Computing the geometry and connectivity codes. The leaves of the kd-tree are sequentially merged following a bottom-up process. For each fusion of two leaves, a code sequence is computed that combines geometry and connectivity information. It should be noted that for rendering needs, this sequence is enriched by specifying the orientation of the triangles that collapse with merging. It is not a complete normal coding, but only a binary description of each triangle orientation. The vertex normals are computed in real-time during the rendering stage by averaging the triangle normals. The process is stopped when the minimal precision of c is reached. In large part, the ideas of [1] are used to compute geometry and connectivity codes, with some adaptations or simplifications to guarantee high frame rates. As for the geometry, each cell subdivision is encoded to describe one of the 3 following events: (0) the first half-cell is non-empty, (1) the second half-cell is non-empty, or (2) both cells are non-empty. The theoretical cost of $log_2 3 = 1.58$ bits per subdivision is reduced to an average of 1.2 bits using entropy coding with per-level statistical data (high levels are composed essentially of 2 codes, whereas low levels contain mainly 0 and 1 codes). As for the geometry, we separate edge expansion from vertex split. Encoding both operations is the same as in [1] (see Sec. 3), with the exception of an additional code describing the orientation of each newly created triangle. Using a simple prediction based on the triangle orientation in the neighborhood, the overcost is made negligible (around 0.01 bit per vertex). It is worth noting that none of the prediction schemes of [1] were used, thus promoting decoding speed and visualization efficiency. However, for a compression-driven application (or in a context where meshes are visualized on high-end computers), the original prediction schemes can definitely be included.

(c) Writing the codes in a temporary file. The codes obtained are written in a semi-compressed temporary file (SCT) using an arithmetic coder [29]. At this stage, we cannot write in the final entropy coded file (FEC) since statistical data are not yet available.

(d) Merging into the parent nSP-cell. We are now left with a kd-tree and a simplicial complex whose vertices have the minimal precision of c. To continue the bottom-up postorder traversal of the nSP-tree, the content of c must be moved to its parent. If c is the first child, its content is simply transferred to the parent. Otherwise, the kd-tree and simplicial complex of c are combined with the parent's current content, which implies detecting and merging the duplicated simplices (see Sec. 7.5).

6.2. Final Output

Once all nSP-cells have been processed, statistical data over the whole stream are available and efficient entropy coding can be applied. The probability tables are added to the header of

the FEC file, then the nSP-cells code sequences of the SCT file are sequentially passed through an arithmetic coder. As geometry code frequencies vary with respect to the level of detail, the probabilities are computed independently for each level. In our case, dynamic probabilities would not improve this static per-level probability scheme since the independence of nSP-cells would require the statistical data to be reinitialized too frequently. At the end of the file, a description of the nSP-tree structure is added that indicates the position of each nSP-cell in the file. This table allows the decompression algorithm to rebuild an empty nSP-tree structure that provides direct access to the code sequence of any nSP-cell.

6.3. Memory and Efficiency Issues

Memory management. To minimize memory occupation, each loaded nSP-cell only stores the leaves of its kd-tree. The internal nodes are easily rebuilt by recursively merging the sibling leaves. Furthermore, the successive refinements and coarsenings imply a great deal of creation and destruction of objects such as kd-cells, simplices and lists. A naive memory management would involve numerous memory allocations and deallocations, causing fragmentation and substantial slowdowns. To avoid this, memory pools are widely used to preallocate thousands of objects in one operation, and preference is given to object overwriting rather than deletion and recreation.

Common implementations of the simplicial complex build a structure where each d-simplex ($d > 1$) is composed of $d+1$ ($d-1$)-simplices. This structure is costly to maintain, and since we do not particularly need to keep this parent-to-child structure, we have chosen to store the following minimal data structure: simplicial complexes are represented as a list of top-simplices, each simplex containing references to the corresponding kd-cells that give a direct link to its incident vertices. Conversely, each kd-cell stores a list of its incident top-simplices.

Finally, the external memory accesses are optimized to avoid frequent displacements of the hard disk heads. As seen in section 5.3, pagination and buffering techniques are used to write in the RWI file. During visualization, each time an nSP-cell is needed, its complete code is read from the FEC file and put into a buffer to anticipate upcoming accesses.

Multi-core parallelization. The nSP-tree structure is intrinsically favorable to parallelization. The compression has been multithreaded to benefit from the emerging multi-core processors. While file I/O remain single-threaded to avoid costly random accesses to the hard disk, vertex splits and unifications, edge expansions and collapses, nSP-cell splits and mergings can be executed in parallel on different nSP-cells. To favor effective sequential access to the hard drive and to avoid frequent locks of the other computing threads (file I/O are exclusive), the processing of an SP-cell starts by loading the list of contained triangles from RWI and RWP files into a buffer. Then the following steps (see Sec. 6.1) only require CPU and RAM operations and can be performed by one thread running parallel to the other threads. Experimentally, a global increase in performance is observed between 1.5 and 2 with a dual-core CPU, and between 2 and 3 with a quad-core CPU. This parallelization similarly benefits the decompression and visualization stage, although the performance increase is more difficult to estimate.

7. Decompression and View-Dependent Visualization

Decompression and visualization are strongly interlinked, since the data must be efficiently decoded and selectively loaded in accordance with the visualization context. In this section, each component of the rendering process is detailed.

7.1. Initialization

The first step consists in reading general informations about the mesh from the FEC file header, then using the offset table situated at the end of the file to build the nSP-tree structure. Once done, each nSP-cell only contains its position in the FEC file. To complete the initialization process, the kd-tree of the root nSP-cell is created at its simplest form, *i.e.* one root, and likewise the associated simplicial complex is initialized to one vertex.

7.2. Real-Time Adaptive Refinement

At each time step of the rendering, the mesh is updated to match the visualization frame. The nSP-cells in the view frustum are refined or coarsened so that the maximum error on the coordinates of any vertex guarantees that its projection respects the imposed display precision. The other cells are coarsened to minimize memory occupation, and are not sent to the graphic pipeline to reduce GPU computing time. According to this viewpoint criterion, a list of nSP-cells to be rendered with their associated level of precision is maintained. A cell that reaches its maximal precision is split, and the content of each child is efficiently accessed thanks to the offset contained in the nSP-tree structure. Conversely, if an nSP-cell needs to be coarsened at a level below its maximal precision, its possible children are merged together.

7.3. Multiresolution Transitions

Once the nSP-cells that need to be rendered are known, two main issues arise, both concerning the borders between nSP-cells. First, the simplices being duplicated in different nSP-cells may overlap. Second, visual artifacts can be caused by adjacent nSP-cells with different levels of precision. Only border top-simplices can cause these problems; the others can be straight rendered using their representative points. Border top-simplices are composed of at least one vertex belonging to another nSP-cell. The following algorithm is applied to each border top-simplex s:

1. Let c be the nSP-cell to which s belongs, p_c the current precision of c, and l the list of nSP-cells to render. Let N be the number of kd-cells (*i.e.* the number of vertices) composing s, c_i (for i in $1, \ldots, N$) the nSP-cell in which lies the ith kd-cell k_i composing s, and p_i the current precision of c_i (if $c_i \notin l$ then $p_i = p_c$).
2. If there is an i in $1, \ldots, N$ such that $p_i > p_c$ or ($p_i = p_c$ and c_i index $>$ c index), s is removed and will not be rendered.

3. Else, for each kd-cell k_i such that $c_i \in l$ and $c_i \neq c$, the kd-cell k'_i of c_i containing k_i is searched. The representative point of k'_i is used to render s.

Thus, a smooth and well-formed transition is obtained between nSP-cells that have different levels of precision (see Fig. 5).

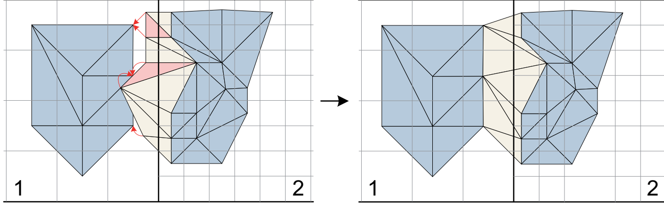

Figure 5: Rendering of the border top-simplices between two nSP-cells with different LODs

7.4. *Encoding Geometry Before Connectivity*

The main drawback of this algorithm is its tendency to produce many small triangles in the intermediate levels of precision. These triangles, which usually have a size near the requested screen precision, slow down the rendering without resolving the well-known block effects inherent to the kd-tree or octree approaches (see Fig. 9, left). This issue is addressed by encoding geometry earlier than connectivity in the FEC file: the decoder will hear of the point splits m bits before the associated connectivity changes. Consequently, a kd-cell c will be composed of a connectivity code that describes the creation of the representative vertex of c, followed by multiple geometry codes that describe the positions of the child vertices of c. The level of the descendants coded in a kd-cell depends on m. For a 3D mesh and a geometry advance of m bits, $3\,m$ levels of children are contained in each kd-cell. For each vertex displayed, its future children are thus known and their center of mass can be taken as the new refined position of the representative vertex (see Fig. 6).

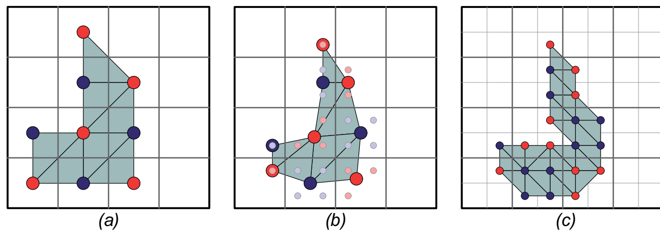

Figure 6: Example of rendering with the geometry advance: comparison between (a) 2 bit precision for both geometry and connectivity, (b) 2 bits for connectivity and 3 bits for geometry (2 + 1 early bit), (c) 3 bits for both

Figure 7 shows an example of an nSP-tree with a geometry advance of 1 bit. During the split of an nSP-cell, some kd-cells are duplicated in several children, such as kd-cell 11 in this example. The duplicates of the same kd-cell may evolve differently, since their neighborhood depends on the nSP-cell they belong to. Thus, their codes may differ, such as G11 and G11b. The parent kd-tree can store the early geometry codes of only one nSP-cell child. We choose the child containing the original kd-cell. For the other nSP-cells, the missing geometry codes are added at the beginning of the sequence. For instance, in our example, nSP-cell #0 stores G11, G23 and G24, corresponding to kd-cells 11, 23 and 24 of nSP-cell #2; missing G11b and G23b are stored at the beginning of nSP-cell #1 (bold red box). Similarly, since the root nSP-cell (#0) has no parents, a few geometry codes must be added at its beginning. Figures 8 and 9 illustrate the benefits of this technique in terms of rendering. The rate-distortion curve, constructed using the *L2* distance computed by the METRO tool [30], shows a significant improvement in the intermediate levels of detail.

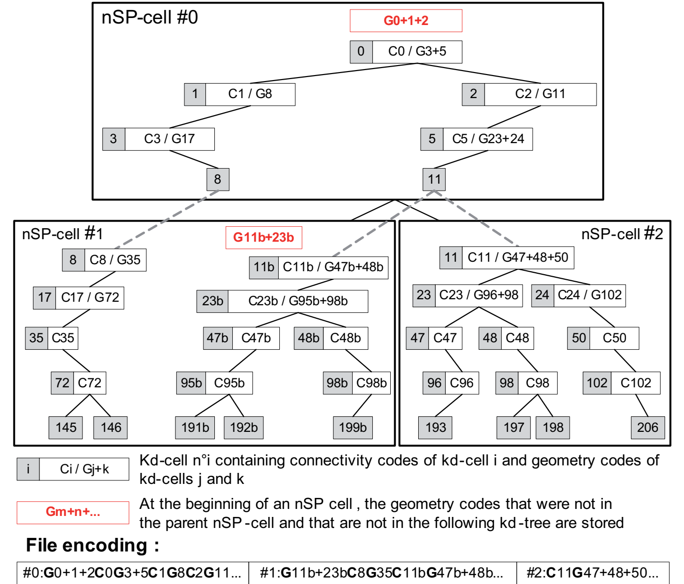

Figure 7: Example of a kd-tree with geometry before connectivity: 2D case, 1 bit earlier (*i.e.* 2 kd-tree levels)

7.5. *Efficient Adaptive Rendering*

To attain good performance in terms of smooth real-time rendering, particular attention was paid to a number of aspects of the implementation. During an nSP-cell split, kd-cells and top-simplices are transferred and sometimes duplicated in the children. A costly part of this split consists in determining the child to which each kd-cell must be transferred and the possible children where it must be copied (when it is incident to a border top-simplex). Rather than computing this information for numerous kd-cells just before the split, we compute it as soon as the kd-cell precision suffices to determine its future containing nSP-cell, store it and transfer it directly to the descendant kd-cells. Likewise, from this information, the progression of a top-simplex during the next nSP-cell split can be deduced. To smooth the computation load over time, each top-simplex is tested either after its creation (due to a kd-cell split) or after its duplication (due to an nSP-cell split).

Conversely, merging two nSP-cells implies deleting all the duplicated objects. To optimize this step and quickly determine these objects, each kd-cell and top-simplex stores the kd-tree level it was created in and whether it was created after an nSP-cell split. Another issue is that top-simplices moved into the

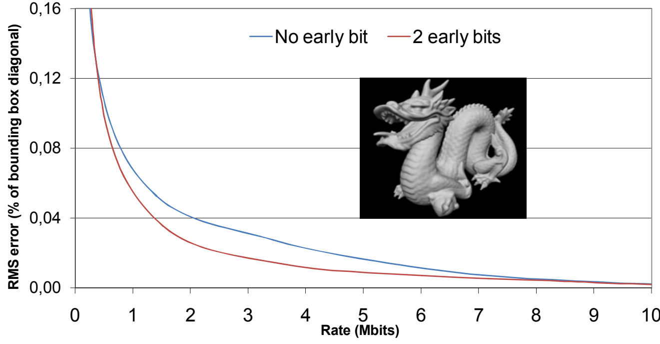

Figure 8: Comparison of rate-distortion curves

Figure 9: Geometry before connectivity: normal rendering (left) and rendering with a precision advance of 2 bits for vertex positions (right)

parent nSP-cell may refer to a duplicate kd-cell that must be deleted. Since we cannot afford to look for the corresponding original kd-cell among all the existing kd-cells, each duplicate stores a pointer to its original cell. To keep this working properly, the coarsening and refinement processes must be reversible: by refining an nSP-cell and then coarsening it back to its minimal precision, the same kd-cells must be obtained.

Computing the connectivity changes associated with kd-cell splitting or merging are costly operations on which we have focused to ensure a fast rendering. First, an efficient order relation is needed over the kd-cells: rather than the geometric position, the kd-index order has been chosen, which costs a single comparison and gives a natural order relation over the top-simplices. In addition, it has been shown that each top-simplex stores the kd-tree level it was created in. This value is also used to speed up edge collapse and vertex unification during coarsening: since it points out the top-simplices that must be removed after merging 2 kd-cells, we only need to replace the removed kd-cell with the remaining kd-cell in the remaining incident top-simplices, then check whether any triangles degenerate into edges.

In order to improve the visualization speed, we implemented prefetching, which can be used either to preload the whole file into memory and thus avoid any subsequent hard drive access (if the file fits into main memory), or to prefetch upcoming nSP-cells from disk by trying to predict the camera position at the next time step. However, this hardly improves the overall performance: for example, the computing time to obtain a view of a close-up on the St. Matthew face is $5.8s$ without any prefetching, whereas it is $5.6s$ with full prefetching (whole file preloaded). This experiment shows that, with a cumulative cost of $0.2s$ over $5.8s$ (3.4%), the loading process is almost negligible. This can be explained by two factors: the first one is that we carefully optimized external data access (buffered and sequential), the second one is that compression drastically reduces the amount of data we need to read. This last point argues in favor of compression, and we can assume that it will become more and more important, in view of the increasing gap between processing units and data access performances [31].

8. Experimental Results

We cannot fairly compare our method to previous work, since, to our knowledge, no other method combines the features of compression and interactive rendering. However, for reference, we provide comparisons to state-of-the-art compression methods (single-rate or progressive, in-core or out-of-core) in Sec. 8.1, then to state-of-the-art visualization methods in Sec. 8.2. The results presented were obtained from a C++/OpenGL implementation of the method running on a PC with an Intel Q6600 QuadCore 2.4Ghz CPU, 4GB DDR2 RAM, 2 RAID0 74GB 10000 RPM hard disks, and an NVIDIA GeForce 8800 GT 512MB video card. Regarding the parameter settings, n must be big enough to allow the selection of small parts of the mesh in a few nSP-cell splits, but small enough to avoid displaying too much nSP-cells simultaneously (in which case borders computation, for instance, could be excessively expensive). A small N_{min} increases the number of duplicated simplices and degrades the compression performance, whereas a big N_{min} leads to a smaller nSP-tree, to the detriment of the multiresolution capability. $n = 4$ and $5000 \leq N_{min} \leq 8000$ appear to be good trade-offs.

8.1. Compression

Table 1 presents results from the out-of-core compression stage. p_r was set at 7 bits so that the number of triangles in the root nSP-cell remains small enough to allow this coarse version of the mesh to be loaded in main memory. For each model, we first indicate the number of triangles, the PLY file size, and the size of the raw binary coding, which is the most compact version of the naive coding: if v is the number of vertices, t the number of triangles and p the precision in bits for each vertex coordinate, this size in bits is given by $3pv$ for the geometry $+3\,t\,log_2 v$ for the connectivity. Then the compression rate is given (as the ratio between the raw size and the FEC size) with the total number of nSP-cells. Compression times are also provided, the column headers referring to the sections that describe the corresponding steps of the algorithm. As expected, the computation of code sequences (see section 6.1) is the most expensive part. However, this step, which is directly proportional to the number of vertices and triangles, benefits from our multithreaded implementation. Finally, the table details the memory usage (including temporary disk usage), attesting that the method is actually out-of-core. It is worth noting that the spatial coherence of the input model is important and can explain, for instance, why the step 6.1 is shorter for *Lucy* than for the *T8* model, where *Lucy* contains more triangles. However, we found that the scanned models we used showed in general a good spatial coherence.

Table 1: Compression results with $p = 16$ bits per coordinates, $n = 4$, $N_{min} = 8000$ and $p_r = 7$

Models	Input file (sizes in MB)			Compressed file		Compression time (mm:ss)				Mem (MB)	
	Triangles	Ply	Raw	Rate	*n*SP-C.	5	6.1	6.2	Total	Ram	HD
Bali (*)	5,435,004	-	33	2.78	10,753	01:43	00:32	00:06	02:22	173	121
Wallis (*)	6,763,447	-	41	3.02	13,505	02:08	00:40	00:07	02:55	174	147
Bunny	69,451	3	0.6	3.37	65	00:01	00:02	00:00	00:03	108	1
Armadillo	345,932	7	3.3	3.95	1,217	00:02	00:03	00:01	00:06	121	6
Dragon	866,508	34	8.7	4.64	1,281	00:07	00:10	00:01	00:18	151	15
David 2mm	8,250,977	173	93	8.05	25,601	0:00:50	0:00:54	0:00:09	0:01:53	290	136
UNC Powerplant	12,388,092	503	174	7.85	38,209	0:04:26	0:01:52	0:00:09	0:06:27	291	290
EDF T5	14,371,932	388	166	7.36	27,521	0:01:28	0:01:50	0:00:16	0:03:34	290	254
EDF T8	22,359,212	693	263	6.82	31,105	0:02:47	0:03:07	0:00:26	0:06:20	289	404
Lucy	27,595,822	533	328	7.35	114,113	0:03:10	0:02:48	0:00:33	0:06:31	299	522
David 1mm	54,672,488	1,182	671	8.81	181,569	0:06:01	0:04:58	0:00:57	0:11:56	304	1035
St. Matthew	372,422,615	7,838	4,686	11.57	353,473	0:40:50	0:33:13	0:06:26	1:20:29	318	6510

(*) These models are unstructured point clouds (the *Triangles* columns gives the number of points)

Table 2: Comparison of compression ratios in bpv

Models	Vertices	p	[32]	[4]	[1]	[28]	[22]	[23]	*CHuMI*
Bunny	35 947	12	-	-	17.8	-	-	-	27.0
Horse	19 851	12	19.3	-	20.3	-	-	-	29.3
Dino	14 050	12	19.8	-	-	31.8	-	-	28.7
Igea	67 173	12	17.2	-	-	25.0	-	-	25.9
David 2mm	4 128 028	16	-	14.0	-	-	306.2	-	22.3
UNC Powerplant	10 890 300	16	-	14.1	-	-	-	-	16.3
Lucy	13 797 912	16	-	16.5	-	-	-	-	25.9
David 1mm	27 405 599	16	-	13.1	-	-	282.3	-	22.2
St. Matthew	166 933 776	16	-	10.7	-	22.9	282.1	508.0	19.4

Table 2 shows the results of our algorithm in terms of bits per vertex compared to those of [32] (a reference in-core single-rate method), the original in-core multiresolution algorithm [1], the single-rate out-of-core method [4] and the single-rate out-of-core random access method [28]. The first three are pure compression methods, while the fourth provides a trade-off between compression and random accessibility, very useful for applications that need mesh traversal but not well-suited to visualization. The two non-compressive visualization methods [22] and [23] have been added for comparison purposes. Unfortunately, we were not able to compare with [17] (to our knowledge, the sole out-of-core and progressive compression algorithm) since the authors only give connectivity rates. Compared to compression methods that do not allow interactive visualization (first three columns), an average extra cost ranging from 15% to 80% is observed, mainly due to the redundancy induced by space partitioning (5—7% of the original triangles are duplicated) and the fact that complex prediction is bypassed to guarantee high frame rates during rendering.

To show that the method behaves well with point clouds (possibly unbalanced) and does not rely on the mesh connectivity, two examples of unstructured models have been added to our experimental pool: *Bali* and *Wallis*.

8.2. Decompression and Visualization

Figure 10 presents a few examples of different LODs of the *St. Matthew* model with the times observed to obtain a given view distance by refinement or coarsening of the previous one (or from the beginning of the visualization for view (a)). They may seem high but it must be noted that during navigation, the transition from one view to the next one is progressive and the user's moves are usually slow enough to allow smooth refinements. This is why the best way to appreciate the algorithm's behavior is to manipulate the viewer or watch the accompanying videos [33]. All the results have been produced with a maximal screen space error tolerance of 1 pixel. To appreciate the data's decoding speed, Fig. 11 presents the number of triangles rendered per second during the *St. Matthew* video. Our decoder can render up to 200 million triangles per second (tps), with an average of about 173 million when it runs at full speed (between $t = 30$ to $t = 65 sec.$ in Fig. 11). For comparison purposes, [20] renders up to 3 Mtps, [22] an average of 70 Mtps, and [23] an

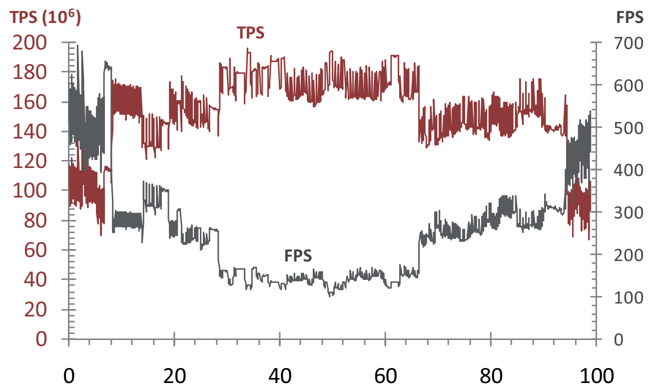

Figure 11: Triangles per second (TPS) and frames per second (FPS) over the time during the accompanying St. Matthew video [33]

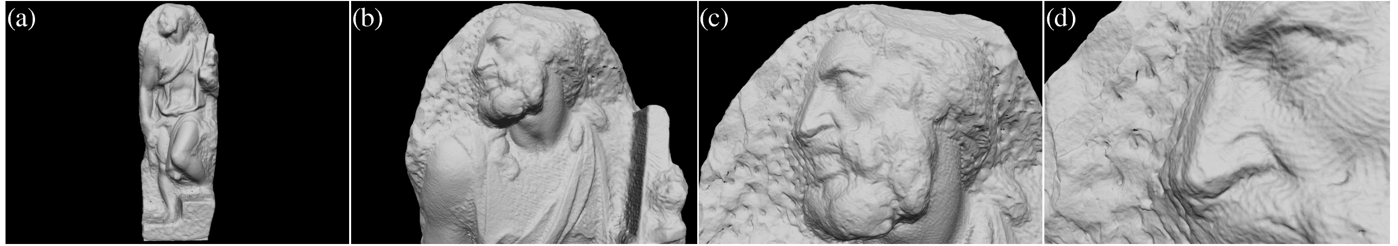

Figure 10: *St. Matthew* refinement times: 0.7 s to display (a), 1.2 s from (a) to (b), 2.0 s from (b) to (c), 1.7 s from (c) to (d); coarsening times: 0.8 s from (d) to (c), 0.6 s from (c) to (b), 0.4 s from (b) to (a).

average of 45 Mtps. Of course, this comparison would not be fair without taking into account each method's year of publication and the progress in hardware performance. In addition, [22] and [23] update the model according to viewing distance and direction, whereas *CHuMI* uses a distance criterion only. On the other hand, it must be remembered that none of these visualization methods achieves compression: for instance, [22] and [23] enlarge the original raw files, respectively, by 10% and 80% on average, as shown in Tab. 2. Fig. 12 shows the number of triangles which are created and deleted per second; zooming in leads to the creation of up to 310,000 triangles per second, while up to 565,000 triangles per second can be destroyed during zooming out. It must be added that the small polyhedral holes that can be observed on the captures and especially on the videos are not artifacts of the viewer but come from the original meshes. It is worth noting that the transitions between the k and $k+1$ levels of precision of an nSP-cell, easily perceptible on the video, occur quite rarely (more precisely, when the zoom factor on the SP-cell is doubled, since one additional bit on point coordinates doubles the rendering accuracy). Consequently, in most cases, if such a transition does not occur during a zoom, this does not mean that the refinement process is stuck but that the threshold is not reached.

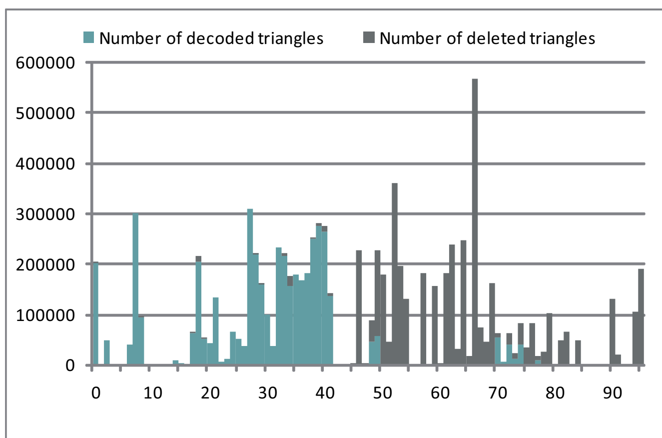

Figure 12: Stacked column chart showing the number of triangles decoded/deleted per second over the time during the St. Matthew video [33]

9. Conclusion and Future Work

As seen in Sec. 8, the data structures and algorithms presented in this article achieve good results compared to the state of the art, at the same time as an out-of-core lossless compression method, and as an interactive visualization method for arbitrary untextured meshes (in terms of type and size), thus offering a valuable combination for numerous applications. However, we are currently improving several points, which should remove the method's main limitations. Our first perspective aims at reducing latency in the navigation stage. To achieve better performance in this respect, at least three paths can be explored: cache-oblivious optimizations, more GPU-friendly data structures such as patch-based LOD, and the intensive use of the GPU via shader programming. Furthermore, we believe that it is still possible to improve the compression ratios by introducing new predictive schemes. Secondly, although theoretically, the method can address meshes in any dimension, some important optimizations must be done to handle volume meshes in good conditions. For instance, the edge collapse and vertex unification operators have to be efficiently extended to tetrahedra. In addition, occlusion culling becomes a major issue in the volume case. Finally, *CHuMI* is based so far on a pure distance criterion to set the precision of displayed triangles. We think that it could be favorably extended to use a hierarchy of surface patch normals and thus enable updates according to view direction as well as distance.

Acknowledgments

The models are provided courtesy of *The Digital Michelangelo Project*, *Stanford University Computer Graphics Laboratory* and *UNC Chapel Hill*. The *EDF Tx* are parts of the *Dancers column* project: *Electricité de France* sponsored the *Dancers column* project for the benefit of *Ecole Française d'Athènes* by offering its technical support and financing the partnerships. The Greek authorities of the *Museum of Delphi* gave access to the column. The *Insight* team was selected for the survey and the registration of the data, with a contribution of *Paolo Cignoni* from *Visual Computing Lab of the Inst. of Information Science and Technologies* of the *Italian National Research Council*. The *Department of Computer Science at University of Bristol* kindly provided a scanner. This research was part of the ACI project *Eros3D* and the ANR project *Triangles*,

supported by the French *Centre National de la Recherche Scientifique* (*CNRS*).

References

[1] P.-M. Gandoin, O. Devillers, Progressive lossless compression of arbitrary simplicial complexes, in: ACM SIGGRAPH Conference Proc., 2002.
[2] C. Gotsman, S. Gumhold, L. Kobbelt, Simplification and compression of 3d meshes, in: Tutorials on Multiresolution in Geometric Modelling, Springer, 2002, pp. 319–361.
[3] P. Alliez, C. Gotsman, Recent advances in compression of 3d meshes, in: In Proc. of the Sym. on Multiresolution in Geometric Modeling, Springer, 2003.
[4] M. Isenburg, S. Gumhold, Out-of-core compression for gigantic polygon meshes, in: SIGGRAPH 2003 Conference Proc., 2003.
[5] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, W. Stuetzle, Mesh optimization, in: SIGGRAPH 93 Conference Proc., 1993.
[6] J. Popović, H. Hoppe, Progressive simplicial complexes, in: SIGGRAPH 97 Conference Proc., 1997.
[7] G. Taubin, A. Guéziec, W. Horn, F. Lazarus, Progressive forest split compression, in: SIGGRAPH 98 Conference Proc., 1998, pp. 123–132.
[8] G. Taubin, J. Rossignac, Geometric compression through topological surgery, ACM Transactions on Graphics 17 (2).
[9] D. Cohen-Or, D. Levin, O. Remez, Progressive compression of arbitrary triangular meshes, in: IEEE Visualization 99 Conf. Proc., 1999, pp. 67–72.
[10] P. Alliez, M. Desbrun, Progressive compression for lossless transmission of triangle meshes, in: SIGGRAPH 2001 Conference Proc., 2001.
[11] J. Peng, C.-C. J. Kuo, Geometry-guided progressive lossless 3d mesh coding with octree decomposition, in: ACM SIGGRAPH Conference Proc., 2005.
[12] P. Lindstrom, C. T. Silva, A memory insensitive technique for large model simplification, in: VIS '01: Proceedings of the conference on Visualization '01, IEEE Computer Society, Washington, DC, USA, 2001, pp. 121–126.
[13] P. Lindstrom, Out-of-core simplification of large polygonal models, in: SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000, pp. 259–262.
[14] P. Cignoni, C. Montani, C. Rocchini, R. Scopigno, External memory management and simplification of huge meshes, IEEE Transactions on Visualization and Computer Graphics 9 (4) (2003) 525–537.
[15] M. Isenburg, P. Lindstrom, L. Livermore, Streaming meshes, in: IEEE Visualization, 2005, pp. 231–238.
[16] M. Isenburg, P. Lindstrom, J. Snoeyink, Streaming compression of triangle meshes, in: SGP '05: Proceedings of the third Eurographics symposium on Geometry processing, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 2005, p. 111.
[17] K. Cai, Y. Liu, W. Wang, H. Sun, E. Wu, Progressive out-of-core compression based on multi-level adaptive octree, in: ACM international Conference on VRCIA, ACM Press, New York, 2006, pp. 83–89.
[18] S. Rusinkiewicz, M. Levoy, Qsplat: a multiresolution point rendering system for large meshes, in: Conf. on Computer Graphics and Interactive Techniques, ACM Press, New York, 2000, pp. 343–352.
[19] J. El-Sana, Y. jen Chiang, External memory view-dependent simplification, Computer Graphics Forum 19 (2000) 139–150.
[20] P. Lindstrom, Out-of-core construction and visualization of multiresolution surfaces, in: Sym. on Interactive 3D Graphics, ACM Press, 2003, pp. 93–102.
[21] S.-E. Yoon, B. Salomon, R. Gayle, D. Manocha, Quick-vdr: Interactive view-dependent rendering of massive models, in: Proc. of Visualization, IEEE Computer Society, 2004, pp. 131–138.
[22] P. Cignoni, F. Ganovelli, E. Gobbetti, F. Marton, F. Ponchio, R. Scopigno, Adaptive tetrapuzzles: Efficient out-of-core construction and visualization of gigantic multiresolution polygonal models, ACM Transactions on Graphics 23 (3) (2004) 796–803.
[23] E. Gobbetti, F. Marton, Far voxels: a multiresolution framework for interactive rendering of huge complex 3d models on commodity graphics platforms, in: ACM SIGGRAPH, ACM Press, 2005, pp. 878–885.
[24] P. Cignoni, F. Ganovelli, E. Gobbetti, F. Marton, F. Ponchio, R. Scopigno, Batched multi triangulation, in: Proceedings IEEE Visualization, IEEE Computer Society Press, Conference held in Minneapolis, MI, USA, 2005, pp. 207–214.
[25] L. Hu, P. Sander, H. Hoppe, Parallel view-dependent refinement of progressive meshes, in: ACM Symposium on Interactive 3D Graphics and Games 2009, 2009.
[26] J. F. Oliveira, B. F. Buxton, Pnorms: platonic derived normals for error bound compression, in: VRST '06: Proceedings of the ACM symposium on Virtual reality software and technology, ACM, New York, NY, USA, 2006, pp. 324–333.
[27] R. Namane, F. O. Boumghar, K. Bouatouch, Qsplat compression, in: ACM AFRIGRAPH, ACM Press, New York, 2004, pp. 15–24.
[28] S. Yoon, P. Lindstrom, Random-accessible compressed triangle meshes, IEEE Trans. on Visualization and Computer Graphics 13 (6) (2007) 1536–1543.
[29] S. Amir, Introduction to arithmetic coding theory and practice, Tech. rep., HP Labs report HPL-2004-76 (April 2004).
[30] P. Cignoni, C. Rocchini, R. Scopigno, Metro: Measuring error on simplified surfaces, Comput. Graph. Forum 17 (2) (1998) 167–174.
[31] J. Hennessy, D. Patterson, Computer Architecture - A Quantitative Approach, Morgan Kaufmann, 2007.
[32] C. Touma, C. Gotsman, Triangle mesh compression, in: Graphics Interface 98 Conference Proc., 1998, pp. 26–34.
[33] C. Jamin, P.-M. Gandoin, S. Akkouche, CHuMI videos (2009).
URL `http://clementjamin.free.fr/CG/`