A Compact Data Structure For The Interactive Navigation Into Arbitrary Meshes

Clément Jamin, Pierre-Marie Gandoin and Samir Akkouche

Abstract

The preprocessing of large meshes to allow and optimize interactive visualization implies a complete reorganization that often introduces a significant data growth. This is detrimental not only to storage or network transmission, but also to the efficiency of the visualization process itself because of the increasing gap between computing times and external access times. In this article, we try to reconcile lossless compression and visualization by proposing a data structure which radically reduces the size of the object while supporting a fast interactive navigation. In addition to this double capability, our method works out-of-core and can handle meshes containing several hundreds of millions vertices. Furthermore, it presents the advantage of dealing with any n-dimensional simplicial complex, which includes triangle soups or volumetric meshes. The main current limit of the data structure lies in its performances in term of real-time visualization: the obtained framerates are usually lower than those of the best dedicated methods, even if the navigation remains practical in the heaviest of all tested models.

Categories and Subject Descriptors (according to ACM CCS):
I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling

1. Introduction

Mesh compression and mesh visualization are two fields of computer graphics particularly active today, whose constraints and goals are usually incompatible if not opposite. Indeed, the reduction of redundancy often goes through a complexification of the signal, 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 is conflicting with the speed requirements of a real-time visualization. Conversely, to allow the efficient navigation through a mesh integrating dynamically the user's interactions, the signal must be carefully prepared and hierarchically structured. This generally introduces a strong redundancy, and sometimes comes with some kind of data loss (in the methods where the original vertices and polyhedra are approximated by some simpler geometrical primitives). Besides the on-disk storage and network transmission issues, the data growth implied by this kind of preprocessing can be detrimental to the visualization itself if the increasing gap between the processing times and the access times from external memory is taken into account.

In this article, we propose to reconcile the two fields by tackling the mesh visualization problem from a pure compression approach. As a starting point, we have chosen an in-core progressive and lossless compression algorithm introduced by Gandoin and Devillers [GD02]. On top of its competitive compression ratios, we propose to add out-of-core and LOD capabilities, in order to handle meshes without size limitation 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 reach these goals, the basic intuitive idea consists in subdividing the original object into a tree composed of independent meshes. This partitioning is realized by introducing a second hierarchical structure (an *n*SP-tree) in which the original data structures (a kd-tree coupled to a simplicial complex) are embedded.

After a presentation of related works (Sec. 2) with a special focus on the compression method chosen as starting point (Sec. 3), our main contribution is detailed through two complementary sections. First, the data structures and algorithms used by the out-of-core compression are introduced (Sec. 4), then the visualization point of view is adopted to complete the description (Sec. 5). In Sec. 6, more details are provided about the implementation strategies that make possible the real-time navigation. Finally, an adaptation of the method is described that appreciably improves the visual quality of the rendering (Sec. 7), before the presentation of commented results (Sec. 8) and the conclusion (Sec. 9).

2. Previous Works

2.1. Compression

Mesh compression is a domain situated between computational geometry and standard data compression. It consists in efficiently coupling geometry encoding (the vertex positions) and connectivity encoding (the relations between vertices), and often addresses manifold triangular surface models. We distinguish single resolution algorithms, which require a full decoding to visualize the object, from multiresolution methods that allow to see the model progressively refined while decoding. Although historically, geometric compression began with single resolution algorithms, we have chosen not to detail these methods here. Similarly, lossy compression, whose general principle consists in working into the frequential domain, is put apart 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 about ten years, and rather than exhaustivity, we have chosen to adopt here some kind of historical point of view. Early techniques of progressive visualization, based on mesh simplification, were not compression-oriented and often induced a significant increase of the file size, due to the additional storing cost of a hierarchical structure [HDD*93, PH97]. Afterwards, several single resolution methods have been extended to progressive compression. For example, Taubin *et al.* [TGHL98] proposed a progressive encoding based on Taubin and Rossignac algorithm [TR98]. Cohen-Or *et al.* [COLR99] used techniques of sequential simplification by vertex suppression for connectivity, combined with position prediction for geometry. Alliez and Desbrun [AD01] proposed an algorithm based on progressive removing 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 [DG00, GD02] 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 triangles soups. Peng and Kuo [PK05] took this paper as a basis to improve the compression ratios using efficient prediction schemes (sizes decrease by about 15%), still limiting the scope to triangular models. Cai *et al.* [CLW*06] 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 in-core progressive algorithms based on octrees.

2.2. Visualization

Fast and interactive visualization of large meshes is a very active research field. The common idea of these works is to select pertinent information to render: the algorithms are told out-of-core, meaning that only necessary and sufficient data at any moment are loaded into memory. A tree or a graph is often built to handle a hierarchical structure. Rusinkiewicz and Levoy [RL00] introduced QSplat, the first out-of-core point-based rendering system, where the points are spread into a hierarchical structure of bounding spheres. These spheres allow to handle easily levels of detail, and to perform visibility and occlusion tests. Several millions points per second can thus be displayed using an adaptive rendering. Afterwards, Lindstrom [Lin03] developed a method for meshes with connectivity. An octree is used to dispatch the triangles into clusters and to build a multiresolution hierarchy. A quadric error metric allows to choose the representative points positions for each level of detail, and the refinement is guided by visibility and screen space error. Yoon *et al.* [YSG05] 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 levels of detail. Cignoni *et al.* [CGG*04] 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 framerates. Gobbetti and Marton [GM05] introduced the *far voxels*, capable to render 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 performances.

2.3. Combined Compression and Visualization

Progressive compression methods are now mature (obtained rates are close to theoretical bounds) and interactive visualization of huge meshes has been a reality for a few 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 low file size to the detriment of a fast data access, whereas visualization methods focus on rendering speed: both goals are opposing and competing. Among the few works that introduce compression into visualization methods, Namane *et al.* [NBB04] developed a QSplat compressed version by using Huffman and differential coding for spheres (position, radius) and normals. The compression ratio is about 50% compared to original QSplat files but the scope is limited to point-based rendering. More recently, Yoon and Lindstrom [YL07] 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 to load the full mesh.

3. Progressive Compression as Starting Point

In this section, we briefly present the method originally proposed by Gandoin and Devillers [GD02] and slightly modified to fulfill our needs. The algorithm, valid in any dimension, is based on a top-down kd-tree decomposition by cell subdivision. The root cell is the bounding box of the point set and each cell subdivision is encoded with $log_2 3$ bits to describe one of the 3 following events: the first half-cell is non-empty, the second half-cell is non-empty, or the two cells are non-empty. As the algorithm progresses, the cell size decreases and the transmitted data lead to a more accurate localization. The subdivision process iterates until there is no more non-empty cell greater than 1 by 1, 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 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 is encoded by symbols inserted between the geometry codes. The vertex merging is performed by the two following decimating operators:

- Edge collapse, originally defined by Hoppe *et al.* [HDD*93] and widely used in surface simplification, is used to merge two adjacent cells under some hypotheses. The two 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) by degeneracy (see Fig. 1a).
- Vertex unification, as defined by Popović and Hoppe [PH97], is a more general operation that allows to merge 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 about twice as big as an edge collapse one.

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).

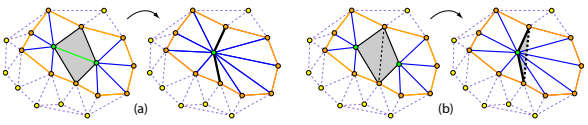

Figure 1: *(a) Edge collapse, (b) Vertex unification*

If this lossless compression method reaches competitive ratios and can handle arbitrary simplicial complexes, it is not appropriate for the interactive navigation through large meshes. Not only its memory footprint makes it strictly impracticable for meshes over about 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 kind of local refinement: indeed, given 2 connected vertices v and w in any intermediate level of detail 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. So, in terms of connectivity, the refinement of the cell containing v cannot be done 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, new algorithms and data structures are presented that are based on the same kd-tree decomposition but remove these limitations.

4. Out-of-core Compression

In this section, we describe in detail the successive steps of the compression process, whose purpose is to code the data in a compact way while restructuring them to make possible a real-time interactive rendering in the visualization stage.

4.1. Partitioning Space

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 get integer coordinates relative to the box, according to the precision p (number of bits per coordinate) requested by the user. These points are written to a raw file (RWP) using a classical raw binary encoding with constant size codes, such that i^{th} point can be directly accessed by file seeking.

Definitions: An nSP-tree is a space-partitioning tree with n splits per axis. Each internal node delimitates a cubic subspace called nSP-cell. Since our meshes are embedded in $3D$, an nSP-cell has n^3 children, and in our case, the nSP-cells are uniformly split, *ie.* their n^3 children always have the same size.

Each nSP-cell c contains vertices whose precision can vary according to the distance from c to the camera. The limit values of the vertex precision into 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, *ie.* 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 kind.

Principle: 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 disk footprint while maintaining a fast and simple reading. Fig. 2 presents a 2D example of such an nSP-tree.

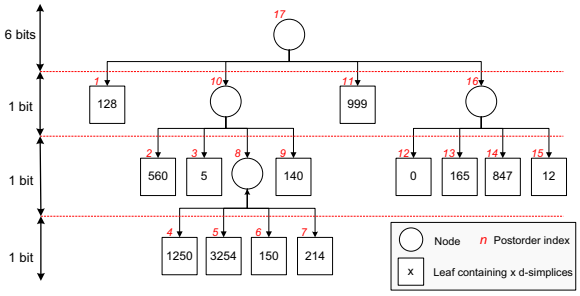

Figure 2: *2D nSP-tree example with $p = 9$ bits, $p_r = 6$ bits, $n = 2$ and $N_{min} = 1000$*

Simplex duplication: A simplex is said to belong to an nSP-cell c if at least one of its incident vertices (0-simplices) lies inside c. Therefore, a simplex of the original model can simultaneously belong to several nSP-cells and consequently, some simplices are duplicated during the distribution into the leaves of the nSP-tree. Fig. 3 shows the space partitioning of an nSP-cell.

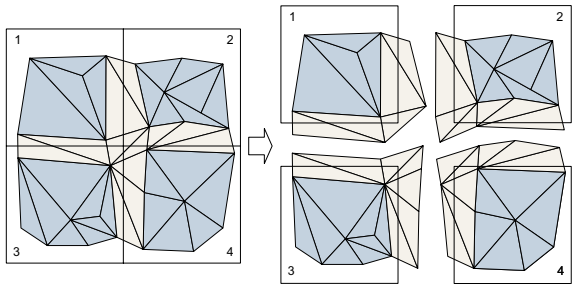

Figure 3: *Example of 2D space partitioning (n = 2)*

Choosing n: In order to get 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 to overlap two nSP-cells.

Writing the header of the compressed file: The file begins with a header that contains all general data required by the decoder: the coordinate system (origin and grid resolution, useful 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 precisions, p_r and p_l.

4.2. Treating each cell of the nSP-tree

Then the nSP-tree is traversed in postorder (see Fig. 2, numbers in red), 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 at last 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 of the kd-tree, a code sequence is obtained that combines geometry and connectivity information. It must be noticed that for rendering needs, this sequence is enriched by specifying the orientation of the triangles that possibly collapse with the merging. The process is stopped when the minimal precision of c is reached.

(c) Writing the codes in a temporary file: The obtained codes are written in a semi-compressed temporary file using an arithmetic coder . At this stage, we cannot write in the final entropy coded file since statistical data over the full compression process is not available.

(d) Merging into the parent nSP-cell: We are left now 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. Else, the kd-tree and simplicial complex of c are combined with the parent current content, which implies in particular to detect and merge the duplicated simplices (see Sec. 6.2).

4.3. Final Output

Once all nSP-cells have been processed, we own statistical data over the whole stream and thus can apply efficient entropy coding. The probability tables are first written to the final compressed file, then the nSP-cells code sequences of the temporary file are sequentially passed through an arithmetic coder. 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 a direct access to the code sequence of any nSP-cell.

5. Decompression and View-Dependent Visualization

The decompression and visualization stages are strongly interlinked, since the data must be efficiently decoded and selectively loaded in accordance with the visualization context. In this section, we detail each component of the rendering process.

5.1. Initialization

The first step consists in reading the header, from which general data is extracted. Then, the offset table situated at the end of the file is read to build the *n*SP-tree structure. Once done, each *n*SP-cell only contains its position in the file. To complete the initialization process, the kd-tree of the root *n*SP-cell is created at its simplest form, *ie.* one root, and likewise the associated simplicial complex is initialized to one vertex.

5.2. Real-time Adaptive Refinement

At each time step of the rendering, the mesh is updated to match the visualization frame. The *n*SP-cells contained by 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, we maintain a list of *n*SP-cells to be rendered with their associated precision. 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 *n*SP-tree structure. Conversely, if an *n*SP-cell needs to be coarsened at a level below its minimal precision, its possible children get merged together.

5.3. Multi-resolution Transitions

Once the *n*SP-cells that need to be rendered are known, we face two main issues, which both concern the borders between *n*SP-cells. First, some of the simplices being duplicated in different *n*SP-cells, they may overlap. Second, visual artefacts can be caused by adjacent *n*SP-cells with different precisions. Only border top-simplices may cause these problems, the other ones can be straight rendered using their representative points. Border top-simplices are composed of at least one vertex belonging to another *n*SP-cell. The following algorithm is applied to each border top-simplex s:

1. Let c be the *n*SP-cell to which s belongs, p_c the current precision of c, and l the list of *n*SP-cells to render. Let N be the number of kd-cells (*ie.* of vertices) composing s, c_i (for i in $1,\ldots,N$) the *n*SP-cell in which lies the i^{th} 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 exists 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 eliminated and will not be rendered.
3. Else, for each kd-cell k_i such that $c_i \in l$ and $c_i \neq c$, we look for the kd-cell k'_i of c_i containing k_i. The representative point of k'_i is used to render s.

Thus, a smooth and well-formed transition is obtained between *n*SP-cells of different precisions (see Fig. 4).

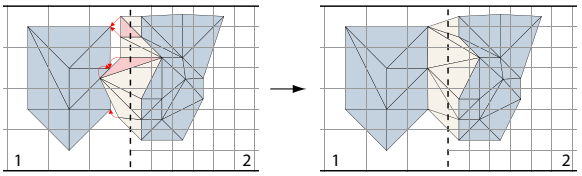

Figure 4: *Rendering of the border top-simplices between two nSP-cells with different LOD*

6. Memory and Efficiency Issues

Two challenges have lead the present work : out-of-core handling of arbitrarily large meshes and smooth real-time rendering. To reach these goals, we had to pay a particular attention to every aspect of the implementation.

6.1. Memory Management

To minimize memory occupation, each loaded *n*SP-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 lot 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 simplicial complex build a structure where each n-simplex ($n > 1$) is composed of $n+1$ $(n-1)$-simplices. Such a 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.

At last, the external memory accesses are optimized to avoid frequent displacements of the hard disk head. Pagination and buffering techniques are used during compression to write in the RWI file, and during visualization to access the compressed file: each time a *n*SP-cell is needed, its complete code is read and buffered to anticipate possible subsequent accesses.

6.2. Efficient Adaptive Rendering

During an *n*SP-cell split, kd-cells and top-simplices are transferred and sometimes duplicated into the children. A costly part of this split consists in determining the child where 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 *n*SP-cell, store it and transfer it directly to the descendant kd-cells. Likewise, from this information, the evolution of a top-simplex during the next *n*SP-cell split can be deduced. To smooth the computation load through time, each top-simplex is tested either after its creation (due to a kd-cell split) or after its duplication (due to an *n*SP-cell split).

Conversely, the merging of two *n*SP-cells implies to delete 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 if it was created after an *n*SP-cell split. Another issue is that top-simplices moved into the parent *n*SP-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 has to be reversible: by refining an *n*SP-cell then coarsening it back to its minimal precision, the same kd-cells must be obtained.

Computing the connectivity changes associated to kd-cell split 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. Besides, we have seen 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 the merging of 2 kd-cells, we just need to replace in the remaining incident top-simplices the removed kd-cell by the remaining one, and to check if some triangles degenerate into edges.

6.3. Multi-core Parallelization

The *n*SP-tree structure is intrinsically favorable to parallelization. Both compression and decompression have 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, *n*SP-cell splits and mergings, can be executed in parallel on different *n*SP-cells. We observe experimentally a global performance increase between 1.5 and 2 with a dual-core CPU, and around 2.5 with a quad-core CPU.

7. Encoding Geometry Before Connectivity

The main drawback of this algorithm is its inclination to produce many small triangles in the intermediate levels of precision. These triangles usually have a size near the asked screen precision and penalize the rendering. Actually, the precision asked by the user mainly concerns the geometry: a one-pixel precision requires the vertex positions to be one-pixel precise but does not require the triangles to be one-pixel sized. This issue is addressed by encoding geometry earlier than connectivity in the compressed file: the decoder will hear of the point splits *m* bits before the associated connectivity changes. It works as if two distinct kd-trees were maintained: the first one would contain geometry split information, and the second one the connectivity codes. The decoding process traverses the first one with an advance of *m* bits over the second one. For each displayed vertex, its future children are thus known and their center of mass is taken as the new refined vertex position. For efficiency purposes, both geometry and connectivity are kept in one slightly modified kd-tree, as can be seen in Fig. 5.

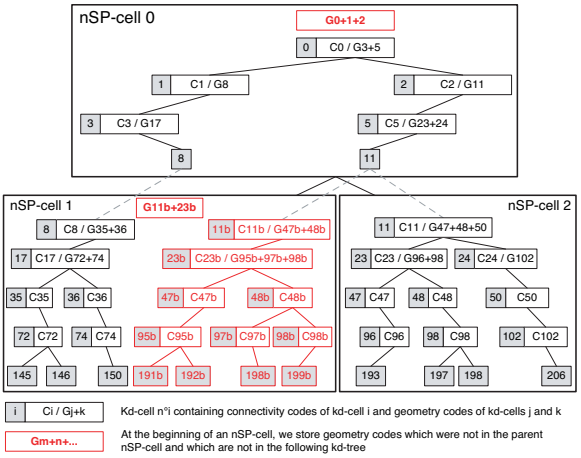

Figure 5: *Example of a kd-tree with geometry before connectivity: 2D case,* 1 *bit earlier (ie.* 2 *kd-tree levels)*

During the split of an *n*SP-cell, some kd-cells may be duplicated in several children, such as kd-cell 11 in Fig. 5. The parent kd-tree (*n*SP-cell 0 in the example) can store the early geometry codes of only one *n*SP-cell child (*n*SP-cell 2). We choose the child containing the duplicated kd-cell. If the kd-cell is outside the parent *n*SP-cell, no child geometrically contains it, so we choose the nearest child. For the other *n*SP-cells, the missing geometry codes are added at the beginning of the sequence (see bold red box in *n*SP-cell 1). Similarly, the root *n*SP-cell having no parent, a few geometry codes have to be encoded at its beginning. Fig. 6 shows the benefits of this technique in terms of rendering.

Figure 6: *Geometry before connectivity: normal rendering (left) and rendering with a precision advance of* 2 *bits for vertex positions (right)*

8. Results

In this section, we present experimental results 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. The tested models are provided courtesy of *The Digital Michelangelo Project*, *Stanford University Computer Graphics Laboratory*, *UNC Chapel Hill* and *Electricité de France (EDF)*.

8.1. Compression

Tab. 1 presents results from the out-of-core compression stage with $p_r = 7$ bits and $n = 4$. For each model, we first indicate the number of triangles and the size of the raw binary coding, which is the most compact version of the naive coding: if n is the number of vertices, t the number of triangles and p the precision in bits for each vertex coordinate, this size in octets is given by $3pn/8$ for the geometry $+3t\,log_2 n/8$ for the connectivity. Then the compression parameters are given, as well as the total number of nSP-cells and of course the compression rate (as the ratio between the raw size and the compressed size). Compared to [GD02], an average extra cost of about 30% is observed, due to the redundancy induced by space partitioning, the additional encoding of triangle orientation, and the fact that we cannot afford to use on-the-fly computed prediction but only precalculated statistical prediction in order to keep rendering interactive. Compression times are also given, the column headers referring to the steps detailed in Sec. 4. As expected, the computation of code sequences (step 2c) 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.

8.2. Decompression and Visualization

Since the refinement and rendering processes run in two different threads, the framerate is not a significant data to evaluate our visualization method. Thus, it is difficult to present quantitative results of the decompression and rendering stage. Fig. 7 presents a few examples of different LOD of the *St_Matthew* model with the times observed to obtain a given view by refinement or coarsening of the previous one (or from the beginning of the visualization for view (a)).

These times can seem high but it must be noticed that during a navigation, the transition from a view to the next one is progressive and the user's moves are usually slow enough to allow smooth refinements. That is why the best way to appreciate the algorithm behavior is to manipulate the viewer or watch the accompanying videos . It must be added that the small polyhedral holes that can be observed on the captures and especially on the videos are not artefacts of the viewer but come from the original meshes.

9. Conclusion and Future Work

As seen in Sec. 8, the data structures and algorithms presented in this article achieves 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 meshes (in terms of kind and size), thus offering an interesting combination for numerous applications. However, we currently work on several points of improvement which should make it possible to rule out the main limits of the method. Our first perspective consists in a noticeable framerate increase in the navigation stage. To reach better performances in this respect, three paths at least can be explored: the introduction of occlusion culling techniques, the optimization of the decompression algorithm (including prefetching strategies and intensive use of the GPU), and the possible recourse to simplification methods to reduce the number of simplices to be displayed in the worst cases. Furthermore, we think that it is still possible to improve the compression ratios by refining the predictive schemes. The difficulty consists in designing a precise probability model whose computing cost does not penalize the visualization.

Acknowledgements

This research takes place into the ACI project *Eros3D* and the ANR project *Triangles*, supported by the French *Centre National de la Recherche Scientifique* (*CNRS*).

References

[AD01] ALLIEZ P., DESBRUN M.: Progressive compression for lossless transmission of triangle meshes. In *SIGGRAPH 2001 Conference Proc.* (2001).

[CGG*04] CIGNONI P., GANOVELLI F., GOBBETTI E., MARTON F., PONCHIO F., SCOPIGNO R.: Adaptive tetrapuzzles: Efficient out-of-core construction and visualization of gigantic multiresolution polygonal models. *ACM Transactions on Graphics 23*, 3 (2004), 796–803.

Table 1: *Compression results*

Models	Input file		Param.		Compressed file		Compression time ([h:]mm:ss)				Mem (MB)	
	Triangles	Raw size	p	N_{min}	Rate	nSP-C.	1	2	3	Total	Ram	HD
Bunny	69 451	609 823	16	5000	3.22	65	00:01	00:01	00:01	00:03	116	2
Armadillo	345 944	3 814 076	24	5000	2.51	1 537	00:05	00:11	00:03	00:19	125	13
Dragon	871 306	10 718 180	28	6000	2.52	1 473	00:18	00:39	00:10	01:07	200	35
UNC_Powerplant	12 748 510	228 106 966	28	8000	3.29	39 489	05:30	10:39	02:23	18:32	178	651
David_2mm	13 797 791	190 286 615	28	8000	3.29	17 217	03:57	08:26	02:10	14:33	149	534
EDF_T5	14 372 492	198 213 076	28	8000	3.21	29 521	04:34	09:24	02:17	16:15	190	567
EDF_T8	22 360 268	313 693 556	28	8000	3.19	31 105	07:57	15:40	03:42	27:19	169	898
Lucy	28 055 742	397 077 691	28	8000	3.13	119 169	08:26	22:43	04:45	35:54	196	1 172
David_1mm	56 230 343	817 791 600	28	8000	3.48	183 169	15:35	36:28	08:48	1:00:51	189	2 274
St_Matthew	372 422 615	5 804 448 412	28	8000	4.17	406 081	1:42:16	2:39:54	52:44	5:14:54	268	14 137

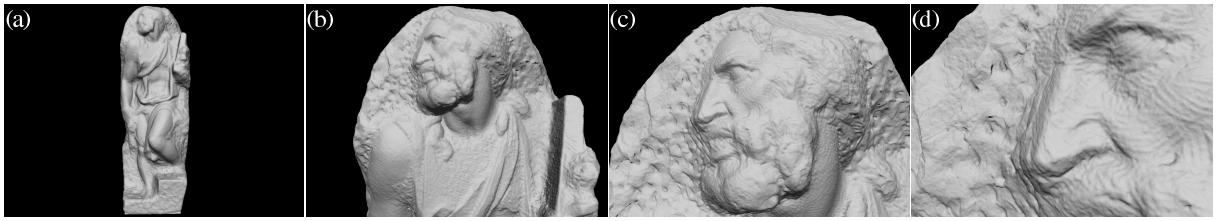

Figure 7: St_Matthew *refinement times: 1.9 s to display (a), 2.7 s from (a) to (b), 2.8 s from (b) to (c), 3.2 s from (c) to (d); coarsening times: 2.9 s from (d) to (c), 2.2 s from (c) to (b), 0.7 s from (b) to (a).*

[CLW*06] CAI K., LIU Y., WANG W., SUN H., WU E.: Progressive out-of-core compression based on multi-level adaptive octree. In *ACM international Conference on VRCIA* (2006), ACM Press, New York, pp. 83–89.

[COLR99] COHEN-OR D., LEVIN D., REMEZ O.: Progressive compression of arbitrary triangular meshes. In *IEEE Visualization 99 Conf. Proc.* (1999), pp. 67–72.

[DG00] DEVILLERS O., GANDOIN P.-M.: Geometric compression for interactive transmission. In *IEEE Visualization 2000 Conference Proc.* (2000).

[GD02] GANDOIN P.-M., DEVILLERS O.: Progressive lossless compression of arbitrary simplicial complexes. In *ACM SIGGRAPH Conference Proc.* (2002).

[GM05] GOBBETTI E., MARTON F.: Far voxels: a multiresolution framework for interactive rendering of huge complex 3d models on commodity graphics platforms. In *ACM SIGGRAPH* (2005), ACM Press, New York, pp. 878–885.

[HDD*93] HOPPE H., DEROSE T., DUCHAMP T., MCDONALD J., STUETZLE W.: Mesh optimization. In *SIGGRAPH 93 Conference Proc.* (1993).

[Lin03] LINDSTROM P.: Out-of-core construction and visualization of multiresolution surfaces. In *Symposium on Interactive 3D Graphics* (2003), ACM Press, New York, pp. 93–102.

[NBB04] NAMANE R., BOUMGHAR F. O., BOUATOUCH K.: Qsplat compression. In *ACM AFRIGRAPH* (2004), ACM Press, New York, pp. 15–24.

[PH97] POPOVIĆ J., HOPPE H.: Progressive simplicial complexes. In *SIGGRAPH 97 Conference Proc.* (1997).

[PK05] PENG J., KUO C.-C. J.: Geometry-guided progressive lossless 3d mesh coding with octree decomposition. In *ACM SIGGRAPH Conference Proc.* (2005).

[RL00] RUSINKIEWICZ S., LEVOY M.: Qsplat: a multiresolution point rendering system for large meshes. In *27th Annual Conference on Computer Graphics and Interactive Techniques* (2000), ACM Press/Addison-Wesley Publishing Co., New York, pp. 343–352.

[TGHL98] TAUBIN G., GUÉZIEC A., HORN W., LAZARUS F.: Progressive forest split compression. In *SIGGRAPH 98 Conference Proc.* (1998), pp. 123–132.

[TR98] TAUBIN G., ROSSIGNAC J.: Geometric compression through topological surgery. *ACM Transactions on Graphics 17*, 2 (1998).

[YL07] YOON S., LINDSTROM P.: Random-accessible compressed triangle meshes. *IEEE Transactions on Visualization and Computer Graphics 13*, 6 (2007), 1536–1543.

[YSG05] YOON S., SALOMON B., GAYLE R.: Quick-vdr: Out-of-core view-dependent rendering of gigantic models. *IEEE Transactions on Visualization and Computer Graphics 11*, 4 (2005), 369–382.