Simplification and Streaming of GIS Terrain for Web Clients

Fabien Cellier[1,2], Pierre-Marie Gandoin[1,3], Raphaëlle Chaine[1,2], Aurélien Barbier-Accary[4], and Samir Akkouche[1,2]

[1]Université de Lyon, CNRS
[2]Université Lyon 1, LIRIS, UMR5205, F-69622, France
[3]Université Lyon 2, LIRIS, UMR5205, F-69676, France
[4]ATOS Worldline, France

Abstract

The application needs in 3D visualization culminate today, in particular in the field of geographic information systems *(GIS)*, as evidenced by the popularity of applications like *Google Earth* or *Google Map*. Meanwhile, the popular success of mobile devices like smartphones or tablets and the explosion of cloud computing directly related to ubiquitous networks accelerates the gradual shift from the traditional desktop application development to web and specialized mobile application development. But if the latest technologies centered around *HTML5* facilitate the development of rich internet applications *(RIA)*, the gap in resources between a desktop computer and a smartphone requires still an important conceptual and algorithmic work when one aims to design web applications offering a user experience similar to desktop applications. In this paper, we propose a method of terrain simplification suitable for data compression and streaming, and therefore ideal for the *GIS* visualization in a web browser. Based on new parallel algorithms, this method was designed to exploit the multi-core architectures of the latest *CPU* and *GPU*, within the constraints of the latest *HTML5 API (WebGL*, *WebSockets*, *WebCL*). It offers the main advantage of working on irregular grids, which allows to modelize highly non-uniform terrains (containing for instance roads and buildings) that may be unprojectable (plain 3D and not only 2.5D).

CR Categories: I.3.3 [Computer Graphics]: Three-Dimensional Graphics and Realism—Viewing Algorithms H.3.4 [Information Storage and Retrieval]: Systems and Software—Information Networks;

Keywords: 3D visualization, Terrain rendering, GIS, Web, WebGL, WebCL

Links: DL PDF

1 Introduction

Geometric data visualization is a field of computer graphics particularly active because of its many industrial applications. Among these applications, the visualization of terrain models occupies a prominent place, as evidenced by the abundant literature on the subject since the 1980s, and the popularity of applications like *Google Earth*. Recent developments in technology — growing power of *GPU* on one hand, new languages and frameworks for efficient web development on the other hand —, offer new possibilities for terrain visualization within web browsers.

e-mail: firstname.lastname@liris.cnrs.fr, aurelien.barbier-accary@atos.net

Indeed, the shift to cloud computing and the latest *HTML5 API* now allow web browsers to be considered as true operating systems. However, *javascript* engines performances are still weak compared to languages like *Java* or *C++*. Therefore, the arrival in browsers of *WebGL* and *WebCL* frameworks, respective ports of *OpenGL* and the *GP/GPU* framework *OpenCL* with performances comparable to their original models, offers fundamental technological complements. Indeed, the combination *HTML5/WebGL/WebCL* makes possible the creation of web applications offering virtually the same functionalities as traditional desktop softwares.

Our initial goal was to develop methods and tools contributing to the creation of a 3D geographic information system (GIS) executable in a web browser. More specifically, we wanted to propose to the *IGN* (French National Geographic Institute) a solution that integrates perfectly with the *Géoportail* [IGN 2012], their present geographic data visualization website, similar to *Google Earth* but with very detailed and diversified data.

It is within this context that we propose here a new method of parallel simplification, compression and streaming for 3D data visualization from geographic information systems. This method is based on the latest advances in both hardware (our algorithms are parallel to exploit the architecture of the latest *CPU* and *GPU*) and software (they have been designed to use the *HTML5 API* built into modern web browsers). Among the advantages of our method, we can cite the handling of irregular meshes containing buildings, roads or other 3D submodels, a consideration of server load problems with the possibility of using static files for streaming, and a number of parallel operations proportional to the size of the mesh.

While there exists a lot of articles on the 3D terrain visualization, most of them consider only one aspect of our needs: that is to say either the quality of visualization, the data compression or the efficient transfer over the network. In this section, we discuss the work that we consider most relevant in relation to our needs, and we invite the reader to refer also to the survey by Pajarola [Pajarola and Gobbetti 2007] to get a more complete view of the domain.

The *Geometry Clipmaps* [Losasso and Hoppe 2004] are based on the hierarchical organization and the manipulation of a fixed grid whose center depends on the point of view. It might be possible to use this method for streaming by adding wavelet compression (a classic image compression tool) and by exploiting the *Geometry Images* [Gu et al. 2002], which allow storage of 3D elements into an image. Despite this, the method does not meet our initial requirements for the following reasons: first, the grid being semi-regular, there is a constant number of triangles on the screen, and these triangles are uniformly distributed, and not according to the information distribution. Moreover, even if this method is very quick and efficient, the resolution of the displayed data depends only on the distance from the camera to the object, making it difficult to set a visible on-screen error below a given threshold s when the rendering conditions do not allow to display multiple triangles in the grid formed by s. For example, to obtain an error less than the pixel size with this method, it would be necessary to display several triangles per pixel.

The *BDAM* [Cignoni et al. 2003] presents an alternative method based on a division of space in binary tree, with constraints on the tree so as not to create cracks. This division generates patches, that is to say areas containing triangles, and the method simplifies pairs of patches by performing edge collapses based on the *QEM* [Garland and Heckbert 1997]. Unfortunately, no solution is proposed to stream data over a network: each data refinement requires a complete reconstruction of the model, which is adapted to the performance of a hard disk on a local machine, but inappropriate for a broadcast through the network.

The idea of the binary tree with constraints was then extended in the *C-BDAM* [Gobbetti et al. 2006; Bettio et al. 2007], in which compression by a *Neuville* wavelet is substituted for the simplification by edge collapse. The wavelet principle is advantageous for the efficiency of its compression and its intrinsic ability to multi-resolution visualization. In order to remove the limitation of *C-BDAM* to 2.5D (thus allowing, for example, the inclusion of small objects into the model), it would be possible to replace the *Neuville* wavelet by that of *Lounsbery* (a 2D surface wavelet). However, although the wavelets and the semi-regular grids are well suited when the representation is mainly composed of low frequencies, they become ineffective in presence of these very high frequencies.

All theses works can be implemented today as web applications, as shown by projects like *OpenScalesGL* [Cellier 2012], a terrain viewer prototype running in web browsers, on smartphones as well as desktop computers, and based on a method derived from Hoppe's *Geometry Clipmaps*. Of course, this still requires some adjustments to reflect the constraints inherent in *WebGL*, the specificities of *javascript* and web browsers and the heterogeneity of clients. But the main problem of these solutions is that they do not handle irregular grids, and therefore do not permit efficient modeling of data containing high density variations, such as terrains with roads, buildings, or other small 3D objects.

In this paper, we propose to couple the binary tree partitioning method of *BDAM* and *C-BDAM* with an original method of parallel simplification that can be done on *GPU* and does not require use of a regular grid. The triangle simplification approach we propose is based (like the $BDAM$) on the quadric error metric, or *QEM* [Garland and Heckbert 1997]. The distance between the original and simplified models at a vertex P is estimated by the sum of the quadratic distances from P to the planes attached to P, namely the planes formed by the triangles of the original model that contain P. This metric has the distinction of being relatively effective in time and space complexity since the quadratic distance to a set of planes can be described by a simple triangular matrix Q_P.

Lindström showed [Lindstrom and Turk 1998] that it was possible to improve these results with two changes: during the fusion of two points, it suffices to consider the planes of the current mesh instead of the original mesh ones; besides, by adding to the simplification process a constraint ensuring a constant volume, the results are more faithful to the original model (regarding the *Hausdorff* metric). However, this simplification is much more costly in terms of time and space complexity and remains sequential, like the *QEM*.

The problem of mesh visualization performed by the *GPU* was also addressed by *Hu et al.* [Hu et al. 2009], but this work does not aim to compute new mesh versions but rather to visualize view-dependent intermediate meshes: the content of these intermediate meshes is computed in a preprocessing step, then the algorithm load them in the *GPU* memory.

Other recent methods use ray casting to visualize terrain models [Dick et al. 2009] or buildings [Cignoni et al. 2007]. However they still require a lot of power on the client side and are only suitable for 2.5D surfaces or for rectangular objects like simple buildings.

In continuation of his initial work, Garland proposed to introduce parallelism into his original *QEM* method [Garland and Shaffer 2002], through a two-step simplification that merges in parallel points that are in the same cell of an octree, before using the sequential simplification of the original method to refine and smooth the last fusions performed in parallel. However, while parallel and implementable on *GPU*, this method is incompatible with data streaming. Indeed, it builds and encodes the transition from an original mesh to a simplified mesh without detailing the intermediate steps, contradicting the concept of streaming, which is based on the coding and transmission of differences between two successive versions of a mesh. A solution would consist in providing the mesh as a regular grid (the octree), then let the client perform the final fusions using the original sequential simplification algorithm. However, this would prevent the handling of irregular triangulations, which is not satisfactory in our application context.

In this paper, we propose to couple the partitioning method of the *BDAM* [Cignoni et al. 2003] to an original algorithm of parallel simplification that can be implemented on *GPU* and does not require the use of a regular grid. Our solution improves the performances of the *BDAM* simplification, adds a compression feature compatible with the out-of-core visualization and the streaming, and offers the ability to handle plainly 3D objects (and not only 2.5D terrain models).

2 Parallel Simplification

Our primary goal was to develop a parallel simplification algorithm capable to exploit the last *CPU* and *GPU* architectures, whose trend for several years is to increase the number of execution cores. Like the *BDAM* and many other methods for 3D visualization, we have chosen to perform a mesh simplification by successive contractions, or *collapse*, of its edges (see Fig. 1). And to control the error induced by these edge collapses, the quadric error metric, or *QEM* [Garland and Heckbert 1997] has been selected.

The *QEM* related to vertex P is represented by the quadratic distance from P to all planes *adjacent* to P in the original mesh M. In the Garland's definition, a plane $\mathcal{P}$ of M is adjacent to a point P of a simplified version of M if $\mathcal{P}$ contains a triangle with one vertex which belongs to the set of ancestors of P, that is to say the points involved in the successive fusions that led to the creation of P.

The first advantage of this metric is that all the information necessary to process a point P, namely the distance to the set of planes formed by the triangles adjacent to P (according to Garland's definition), may be stored in the form of a triangular matrix Q_P of size 4×4, that is to say 10 coefficients by vertex only. This is true at any level of the mesh hierarchy thanks to the following remarkable property: the result of the fusion of points P and P' whose matrices are respectively Q_P and $Q_{P'}$ is a point P'' such that $Q_{P''} = Q_P + Q_{P'}$. By its nature, the *QEM* does not depend on the order in which the points are merged. This greatly facilitates parallelism, since the impact of a suboptimal fusion can be bounded. Indeed, thanks to this metric, the point resulting from a fusion that generates significant errors will naturally be penalized for future fusions with respect to points produced by fusions whose associated error is smaller. This property is fundamental because the parallelism we have introduced requires to perform edge collapses in which the error is not always minimal throughout the whole mesh but only locally, as we shall see a little further.

It should be noted that Lindström, as discussed in the previous section, has proposed a method [Lindstrom and Turk 1998] that results in better quality simplified meshes but is not adapted to parallelism. Indeed, this solution is based on a conservation of the volume between the different versions of the mesh, which implies in particular to include in the error the volume difference between a model and its simplified version. Thus, the same simplified model may have different error values depending on the fusion order, which is contrary to the properties needed by a parallel algorithm. It would be possible to change the Lindström's method to make the results independent of the collapse order, but this would require to store additional information on each edge, in the form of a matrix similar to Q_P. This would involve a substantial additional memory cost, making the method impractical for very large meshes.

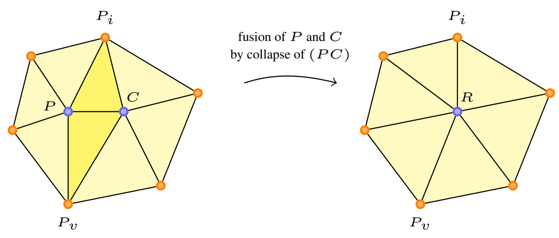

Figure 1: *Example of a reversible fusion between P and C. We define the 1-ring of the fusion as the union of P and C 1-rings minus P and C themselves (ie. the vertices in orange in this figure). The position of R may be set to P or C or another position depending on Q_R.*

The solution we propose is derived from two observations: first, as we have seen, the order in which the points are merged do not influence the error metric, and second, the influence of an edge collapse does not exceed the 1-ring of the fusion, *ie.* the immediate neighbors of the two merged points.

To achieve the parallel fusions, we use four arrays whose respective elements are instances of the following data structures:

```
typedef struct { // 1. Vertex ///////////////////////////
  float x, y, z; // coordinates

  // index of the incident edge that is candidate
  // for the fusion:
  unsigned int candidateEdge;

  // index of one triangle incident to the point,
  // to access the 1-ring:
  unsigned int triangle;

  // initial valence (an integer for padding):
  unsigned int valence;

  // QEM matrix coefficients:
  float a, b, c, d;
  float    e, f, g;
  float       h, i;
  float          k;
} Vertex; // 16 x 4 bytes per point

typedef struct { // 2. Index ////////////////////////////
  unsigned int vertex; // vertex index

  // index which refers to the vertex that stores the
  // split description:
  unsigned int child;

  // used to reconstruct the binary tree and to ensure
  // the same order during simplification and refinement:
  unsigned int parent;

  unsigned int padding; // for padding only
} Index; // 4 x 4 bytes per point

typedef struct { // 3. Triangle /////////////////////////
  unsigned int a, b, c; // vertex (and not index) indices

  // edge indices (edgeX is opposite to vertex X):
  unsigned int edgeA, edgeB, edgeC;

  unsigned int padA, padB; // for padding only
} Triangle; // about 8 x 4 x 2 bytes per point

typedef struct { // 4. Edge /////////////////////////////
  unsigned int trgA, trgB; // adjacent triangles

  float error; // error generated by the edge collapse

  // position of the resulting point:
  float rX, rY, rZ;

  unsigned int padA, padB; // for padding only
} Edge; // about 8 x 4 x 3 bytes per point
```

Coordinates of the vertices are stored together with a reference to the incident edge that could be a candidate for the collapse. The vertices, edges and triangles also correspond to a description of the surface with a direct access to the 1-ring from the vertex. We have also added an index structure used as a vertex handle containing the binary tree generated by the collapse and allowing to work with pointers rather than vertex instances. Note that these data structures have been designed to handle manifold meshes, with or without boundaries.

The *valence* field of the *Vertex* structure is used when the point comes from a fusion of vertices (by edge collapse). It stores the valence of the new vertex, *ie.* its number of neighbors just after the fusion. As discussed in the next section, this value is fundamental for the decompression algorithm.

The array of indices (instances of the $Index$ structure) has a dual purpose. It first allows to exchange element positions without copying all the information associated to a point, which is useful for the sorting step detailed after the algorithm description. In addition, it describes, through the *child* field, the binary tree structure associated with the simplification algorithm. Note that only one *child* field is necessary because the second child is stored in the *Vertex* structure of the non-prioritary merged point (the point C in our notation, see Fig. 1), with other information needed by the decompression and detailed in the next section. Finally, it allows, through the *parent* field, to easily differentiate deleted points from retained ones.

The *Edge* structure is used to store the error associated with each fusion, and the resulting position of the point. It accelerates the search of the best candidate for a given edge collapse, avoiding systematically recalculating each matrix Q_P and retesting each edge after any fusion. The size of the structures is padded to reach the smallest multiple of 16 bytes, thus ensuring an alignment adapted to the aimed hardware architecture, namely the *GPU*, which operates on 4 float arrays for 3D applications. A further point to note is the absence of pointer in our structures: the goal is to be consistent with the upcoming *WebCL* standard which will probably prohibit the use of pointers for security reasons.

With these four arrays, the simplification is done according to the following algorithm:

- Initialization :
    1. In parallel, for each point: the initial matrix Q_P, that describes the equation of quadratic distance to the planes incident to P, is computed.

2. S, the maximum error threshold allowed for a fusion, is fixed (typically, S is user-defined)

- While the number of points in the model exceeds the target number of points:
  - If the number of fusions is 0: the threshold S is increased
  - Else:
    1. Simplification:
       (a) In parallel, for each point P: the index of edge E incident to P whose collapse would generate the minimal error is stored (or *infinite* if this minimal error is greater than S)
       (b) In parallel, for each point P: if P and C are reciprocal candidates for a fusion, they switch to the state "fusion requested"
       (c) In parallel, for each point P: If P requests a fusion (with C) AND no neighbors of P (other than C) request a fusion with a smaller error AND P has priority over C, *ie.* $index(P) < index(C)$:
          i. The coordinates of P are replaced by those of R (the point resulting from the fusion) in the *Vertex* array
          ii. Q_P is updated with $Q_P + Q_C$
          iii. The information allowing to reverse the fusion is stored in Q_C (see Note 1)
          iv. For each triangle T in the 1-ring of the fusion:
             - If T is incident to only one merged vertex: the informations of T are updated (in the *Triangle* array)
             - Else, if T is incident to two merged vertices: T is deleted by setting its vertex indices to -1 (see Note 2)
          v. The 1-ring of the fusion is traversed to update the *Edge* array for the edges incident to the new vertex
          vi. The *Index* array is updated, namely the *child* field of the index corresponding to P and the *parent* field of the index corresponding to C
    2. Sort: the *Index* array is sorted in parallel (bitonic sort), in order to ensure an identical state during simplification and refinement stages.

Notes :

1. To reverse the fusion of the two vertices P and C, it is necessary to store the following information: the positions of P and C, their valences, the indices of P_i and P_v, which constitute with P and C the two triangles that collapse in the fusion (see Fig. 2), and a configuration bit. In addition, it is necessary to store the position in the *Index* array of the element corresponding to P. We will see in detail in the next section how this information is used to perform the reverse operation, *ie.* the vertex split. Besides, since the point C is removed from the mesh, its $Vertex$ and $Edge$ data structures can be used freely for this storage.

2. Note that if the triangle T is incident to a vertex involved in a collapse, then all its vertices belong to the 1-ring of the fusion complemented by the points C and P. But since the 1-ring of a fusion is blocked by our algorithm and cannot merge, it is impossible that two vertices of the same triangle would merge separately. Consequently, the 2 points have necessarily merged together and T collapsed.

The sorting step is necessary to ensure an identical vertex position in the memory during the coding and the decoding. It also helps to separate the retained points from the removed ones after the simplification. For this, the *Index* array is ordered by the index of the parent *(ie.* the point R, result of the fusion, but parent for the split), then by the index of the vertex itself. At the end of this parallel sorting, there exists only one element in the array that has no parent *(ie.* that has not yet been merged), and whose next element in the array is a removed point. We will call this element the pivot of the array. Note that the intrinsic design of the bitonic sort [Batcher 1968], which handles pairs of elements in parallel, allows to save a traversal of the *Index* array: indeed, the pivot position, and therefore the number of points still present in the mesh after the simplification, is detectable at no additional cost during the final step of the sorting.

Note that the algorithm detailed above is nothing but a parallel design of the following process: for each point P, we look into its 2-ring without boundary edges whose collapse results in the minimal error e such that $e < S$. In fact, it is also possible to combine the steps (a), (b) and (c) of the algorithm in a single step. To do so, it suffices to ensure that no point in the 1-ring of P has an apparent possibility of fusion with a smaller error. In the event that such a rival fusion seems to exist, it is not possible to verify that the points P and C' are really reciprocal candidates, since the step (b) has not been done previously. So it can happen in this case that some fusions are rejected unnecessarily, which slightly degrades the performances of parallelism, without jeopardizing the stability of the algorithm, which is guaranteed by the notion of priority.

It is also worth noting that all the update computations are done by the process attached to P, while at first glance it might seem reasonable to let some computations on the process attached to C (the non-prioritary candidate to the fusion which is then removed). However, in the presence of a first parallelism at the point level, it did not seem advisable to add complexity to the algorithm by splitting the update into two asynchronous processes.

We observed experimentally that the simplification of a flat surface modelized by a regular grid was only 3 times faster than the sequential implementation of the *QEM* [Garland and Heckbert 1997]. This can be explained by the fact that a point P can have in this case several equivalent candidates *(ie.* candidates holding the same error value) for the fusion. Then, a random or arbitrary choice of the candidate may prevent some fusions, however legitimate, as shown in Fig. 2. In many methods, in case of candidates of equal priority, it is customary to choose the one that will produce the best quality triangles. However, this criterion would require to compute all the fusions, and moreover, it would not decide in the particular case of a regular grid. Instead, we set up a subgrid G, giving priority to fusions within the same cell of G, which artificially increases the number of configurations similar to that of Fig. 2: each point P chooses its candidate according to the associated errors, then, between tied candidates, the priority will be given to the point that lies within the same cell as P and whose coordinates are minimal. Thus, during the parallel fusion of points in a plane for instance, each point will tend to choose its top left neighbor, except the points lying at the top left corner of a cell of G (like the point D in Fig. 2), for which the fusion will take place. Thus, by defining the subgrid G, the number of parallel fusions is increased: one moves from a unique fusion for an entire area, to a fusion for each cell of G.

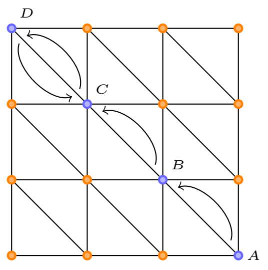

Figure 2: *In this planar regular grid, the fusion of B and A results in the same error as the fusion of B and C. In this case, one can for instance impose to select the candidate with minimal coordinates: B chooses C and A chooses B. In this configuration, only C and D are reciprocal candidates and thus merge, because D is at the top left corner of the plane. To limit these cases that degrade the efficiency of the parallelism, we have implemented a subgrid which artificially increases the number of top left points.*

It should be noted that this parallelization involves fusions based on a local minimum and therefore generates suboptimal results. Indeed, for the same maximum error (the threshold S), we obtain a simplified mesh containing more points than with global minima. However, as mentioned earlier, the *QEM* metric is not sensitive to the fusion order, and the errors of the suboptimal fusions do not add up. Thus, our parallel simplification yields a result that has only 10 to 15% extra points compared to the original method, for execution times about 10 times lower than those obtained with the reference implementation *(QSlim)*, on a computer with an *Intel Core 2 duo 9200 CPU* (2.8Ghz, 2 cores) and an *Nvidia Quadro FX 2700M GPU*.

In some cases, it may be desirable to stop the simplification when a target number of points is reached. But knowing the number of points deleted in the sorted *Index* array is equivalent to know the position of the first point deleted (next to the pivot), hence the importance of the sorting performed at the end of each simplification step.

One drawback of our solution is that simplifications need to be done on all the mesh simultaneously. In fact, one modification to the mesh would introduce new minima wich will change the tree created by the algorithm. Nevertheless, some meshes have too many simplices to be processed in memory, that is why we use the same tiling as performed by the BDAM.

Finally, the complexity of our solution is $n * log^2 n$, where n is the number of vertices, thanks to the bitonic sort. So each parallel task has a complexity of $n * log^2 n$ when the sequential algorithm has a complexity equal to the size of the heap that handles the simplification, namely $n * logn$. We could obtain better performances by replacing the bitonic sort with a radix sort. However, since the radix sort is not stable, the algorithm would no longer be well adapted to streaming.

The figure 3 shows the result obtained by the original sequencial *QEM* simplification method, compared to our parallel algorithm. The visual differences between the two versions of this 3D model are not significant, as is the case for most smooth models.

The figure 5 illustrates the influence of the error treshold on the final simplified mesh. This parameter defines the maximum error that is tolerated during the simplification, which allows the user to specify the accuracy of the resulting mesh. Here, the model is a digital elevation model of 2 millions vertices and the simplification has been stopped at 500k vertices. On the top of the figure, no maximum error has been set and the result is obtained with only 10 successives simplification steps. On the bottom, the treshold has been set to zero and the algorithm needs 18 iterations. We can observe that in this case, the algorithm discards almost all collapses outside the sea to preserve the accuracy of the model. This treshold is useful in particular when a single simplified version of the model is needed: setting the maximum error to a high value yields more regularity in terms of point density and increases the efficiency of the parallelism during the simplification.

3 Compression and Streaming

The parallel simplification algorithm we just presented is an essential element of our method for terrain compression and progressive visualization. It must now be inserted into a reversible compression process compatible with the concept of streaming. In addition, for application purposes, it is fundamental that the refinement process also benefits from parallelism. Indeed, while simplification is generally performed once only and can therefore be handled by thick clients, the refinement, on the contrary, will be typically performed on web thin clients.

As shown by Hoppe [Hoppe 1996], a mesh simplification by edge collapse is reservible. Thus it is always possible to recover the original mesh from a version simplified by our method. However, in order to preserve the original mesh connectivity, two requirements must be met: 1. the order in which fusions that are dependent are performed during the simplification must be precisely reversed during the refinement; 2. the split of R can be performed only if all the neighbors of R *(ie.* the points in its 1-ring) are the same as at the time R was created, at the end of the corresponding fusion.

In our case, the only operation performed during the refinement is the vertex split. This is why a point can never see its valence decrease but only increase, each time one of its neighbors is split in turn. Therefore, only the valence v_R of a point R at the time of its creation (after a fusion) is required by the decoder: during decompression, a point R will be candidate to a split when its valence reaches v_R. Although our algorithm allows to compute and display as much as desired intermediate states, we chose to move from a coarse representation to a finer one by batch, approximately doubling the number of points at each stage, thus leaving aside the possibility of a continuous refinement of the model. This increases the volume of information to be transmitted and therefore the compression efficiency. Of course, this principle of batched point splits is possible only if the two constraints mentioned above are observed.

The advantage of using the valence is that it allows the viewer to identify in parallel points that are candidates for a split: these candidates are all the points R whose current valence is equal to the valence v_R they had at the time of their creation. Note that regarding the compression, the valence distribution, close to a Gaussian distribution, is well suited to an entropy compression of the deviations from the median.

However, to restore the original connectivity between the points after a split, it is also necessary to know the two edges that will form the two new triangles (represented by (RP_v) et (RP_i) in Fig. 1), and the way P and C are connected to the triangles of the R 1-ring, in order to disambiguate the configurations described in Fig. 4.

Regarding the model geometry, the positions of the two points P and C created by a split will be expressed from the position of the parent point R. It is also possible to constrain the position of point R to the set consisting of the two merged points P and C complemented by their middle point. In addition to the benefits gained

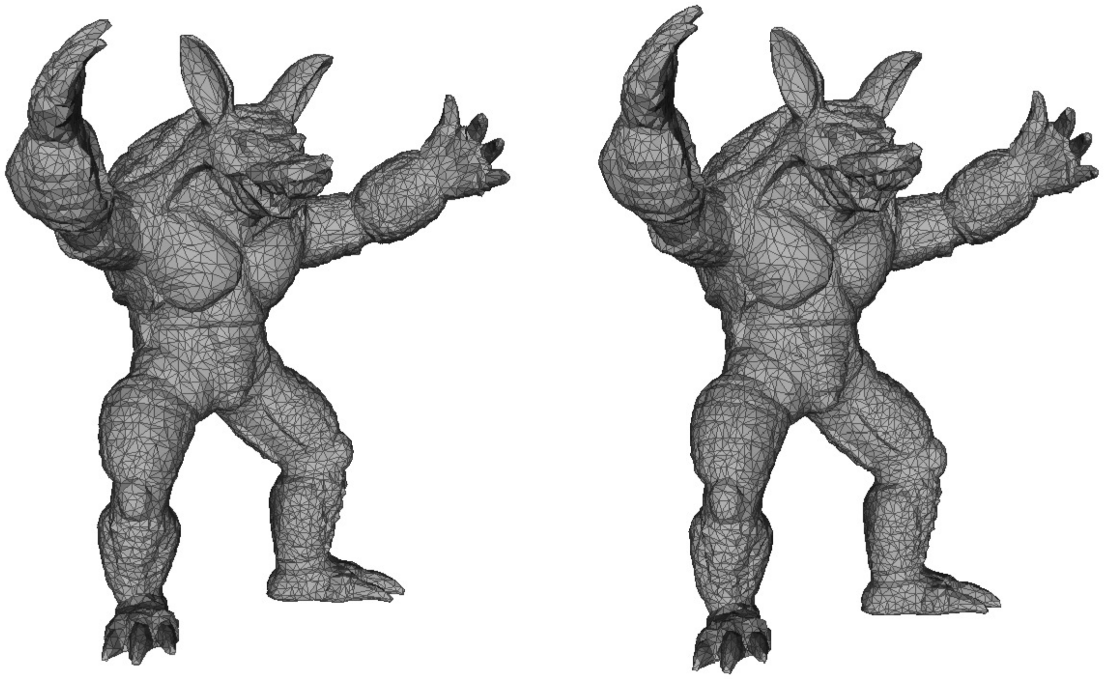

Figure 3: *On the left, a 3D mesh obtained by the sequential simplification algorithm; on the right, by our parallel method.*

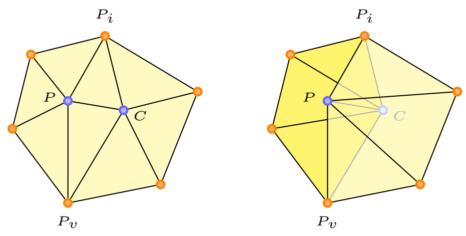

Figure 4: *Two possibilities for a point split. Without a disambiguation bit, it is impossible to choose between these two configurations during the refinement phase. (Note that if the mesh is manifold, only these two cases are possible.)*

in terms of entropy compression, we observed that in the case of buildings or other simple geometric shapes, this constraint generates very few errors. Of course, this observation is less relevant for the terrain models themselves where it is preferable to transmit less points but with a more precise position. Therefore, in the latter case, it is better to choose an optimal positioning to reduce the number of points transmitted and therefore the number of triangles to display for a fixed error treshold and a given volume of transferred data. So the choice of using the optimal positioning or the constrained one depends on the model and will be set for the whole simplification.

Ultimately, to reverse each fusion, the decompression algorithm needs the following information:

- During decompression, P and C will be generated from R. It is therefore natural to transmit the positions of these two points from that of their common parent. So the respective distances of P and C to R are stored: 2×3 floats for a precise positioning, or, in case of constrained positioning, 3 floats (x, y, z) plus a bit specifying whether R is the middle of (PC) or coincides with one of the two points;
- Similarly, to encode the respective valences for which the two new points P and C will become in turn candidates for a split, we store the respective differences from the R valence, *ie.* two integers DV_P et DV_C;
- The indices of edges P_i and P_v among the 1-ring of R;
- One bit to specify the way P and C are connected to the triangles of the 1-ring of R (see Fig. 4).

It is recalled that, in our implementation, all this information is written into the memory space previously dedicated to the matrix Q_C of the deleted point. Indeed, during the fusion, the point P whose index is minimal remains and is updated with the result of the fusion (the point R). The other point, C, is removed from the mesh and updated with the information necessary to perform the split of R. This information could be stored in the *Vertex* structure only, but to save a few casts, it is distributed between the *Vertex* and *Edge* structures as follows:

```
typedef struct { // 1. Vertex //////////////////////////
  // difference between R and P (to retrieve P position):
  float x, y, z;

  // edge index, provides extra integer storage:
  unsigned int candidateEdge;

  // index of the previous child of P:
  unsigned int triangle;

  unsigned int valence; // not used

  // old QEM coefficients,
  // a is used to specify configuration (see figure 3):
  float a, b, c, d;
  float    e, f, g;
  float       h, i;
  float          k;
} Vertex;
```

```
typedef struct { // 4. Edge ////////////////////////////
  // differences between split valences:
  // trgA stores v(P) - v(R) and trgB stores v(C) - v(R)
  unsigned int trgA, trgB;

  float error; // not used

  // difference between R and C (to retrieve C position):
  float rX, rY, rZ;

  // indices of the edges Pv and Pi among 1-ring of R:
  unsigned int padA, padB;
} Edge;
```

Naturally, in order for the streaming to work, it is necessary that the order of points during the refinement on client side be identical to that of the simplification. To ensure this consistency, we have chosen to reorder the points after each simplification stage. Once the array is sorted, removed points are located at the end. Thus, the simplification algorithm can be repeated with all elements of the array from the indices 1 to p, p being the pivot, *ie.* the last point to be kept by the simplification batch.

When the fusions are completed, a second treatment is performed on the *Index* array to create binary files describing the decompression of the mesh, which in turn are compressed by an entropy algorithm (for instance the *gzip* or *deflate* algorithms provided by the *HTTP* protocol). This second step has two objectives:

- Specify the information that the client cannot deduce alone: identify the points that reached their split valence but for which the split is not desirable as it would introduce too little information. One bit for each candidate for a split is necessary to specify whether the split should be done or not.
- Define the granularity of the decompression steps: indeed, it is possible to achieve continuous decompression or, conversely, to define split batches. Limiting the number of batches increases the amount of information sent over the network for each query (or stored in a file), and therefore increases the compression efficiency.

After the entropy decompression which opens each visualization step, it is of course possible to update the mesh in parallel. In particular, in the case of web viewers, and with the arrival of *WebCL*, the parallel version of the update is essential since it allows to overcome the poor performance of *javascript* compared to native languages such as *C*, even when the device has moderate *GPU* power.

However, for a *C++* client application, with a *Core 2 duo 9200M CPU*, a *Nvidia 2700M GPU*, and triangle buffers of 32k points (the maximum size for a good compatibility with *WebGL*, whose current limit is 64k points), it appeared that performing the operations sequentially was the most effective solution in terms of speed and memory management of the *GPU*. This observation can be explained from the current limitations of *GPU* regarding the speed of data transfer from the *CPU* memory, and will probably be no longer valid as soon as *WebGL* accepts large buffers.

Finally, the decompression algorithm running on the client during the mesh refinement can be written as follow:

1. In parallel, for each point R: if the valence of R is equal to the split valence v_R:
   - The point R is added to the buffer of candidates. This operation being parallel, the buffer of candidates will not be ordered. However, it is worth noting that, although the addition is done in parallel, knowing the memory pointer in the candidate array and ensuring its unicity requires a synchronous function *(atomic_inc* in *OpenCL*).
2. The candidates are sorted in parallel (bitonic sort) according to their indices, to ensure that their order is the same as in the simplification stage.
3. In parallel, for each point R candidate for a split: If R holds an error too large (this information is read in the stream), then the split is performed:
   - The two points P and C created by the split are added to the mesh (with their own respective split valences v_P and v_C)
   - The two new triangles (P_iPC) and (P_vCP) are added to the mesh
   - The 1-ring of R is traversed to update the triangles by replacing R with P or C

Model	Size of the connectity information	Size per vertex
France (resolution 1km)	1.90 Mo	15.2 bits
Asian Dragon	2.08 Mo	16.6 bits
Armadillo	1.97 Mo	15.8 bits

Table 1: *Experimental results for the size of the mesh connectivity (ie. the whole coding sequence without the point placement information). For each model, five levels of detail are coded: 1M, 500k, 250k, 125k and 62500 vertices.*

Currently, we face the same problem as *Hu et al.* [Hu et al. 2009]: in practice, the addition of points in the buffer of candidates is performed sequentially. However, as Hu et al., we remain confident that rapidly, the mechanisms provided by the *GPU* should resolve this particular problem. (It is interesting to note that our compression method can also be useful in conjunction with the work of *Hu et al.* to allow an out-of-core progressive visualization of 3D models through the network.)

In terms of compression, the experimental results obtained by our method for the connectivity costs are shown in Tab. 1. When we compare these results with the state of the art, there is still place for improvement: the best mesh compression methods obtain around 15 bits per vertex including the geometry information, as showed for instance in the comparative results of [Jamin et al. 2009]. Nevertheless, those methods address only compression: using a visualization algorithm based on triangle strips is far less efficient regarding the execution times and do not provide the multiresolution. On the contrary, the compression methods that take into account visualization aspects obtain rates around 30 bits per vertex or more, which is comparable to our results.

4 Conclusion and Perspectives

We have presented a parallel algorithm to simplify irregular meshes, which is compatible with streaming and compression. This method, used in conjunction with the partitioning scheme in binary tree proposed in the *BDAM* [Cignoni et al. 2003], provides a complete visualization solution offering a configurable number of intermediate states, particularly suitable for geographic information systems, and implementable in traditional desktop applications as well as in a web browser. This solution is based on local consideration and therefore reduces the information needed to refine the simplified meshes. Indeed, the stream from the server to the viewer consists of a minimal compact data structure that allows the client, after decompression, to reconstruct in parallel the original model.

Among the perspectives of our work, the parallelization of the streaming is prominent. Indeed, since our solution uses entropy coding, it is necessary, each time a patch data has been read over the network, to decompress the stream sequentially before tackling the parallel refinement. This constraint constitutes an important bottleneck for the method, and it would be interesting to work on the possibility to parallelize the decompression step, and maybe the data transfer itself.

Another improvement would consist in replacing the entropy compression of the *HTTP* protocol by an arithmetic encoder driven by a statistical modeller adapted to our output format and data. Nevertheless, it shall be ensured that this optimized encoder does not have a significant impact on decompression times.

In all our treatments, we have chosen to use the segmentation of the *BDAM* method to cut the model into patches easily stored and transmitted over the network. The major issue of this principle of tiling is to prohibit certain fusions in order to avoid the creation of cracks in the mesh. Therefore, the choice of patch size and position influences significantly the model simplification. In our next work, we will study the possibility of a segmentation method that respects the topology and affects as little as possible the choice of merged points.

In the near future, we also wish to address the problem of texture handling, to include them effectively in the current method, both in terms of compression and streaming.

Finally, the solution presented in this paper is not compatible with today's lighter mobile devices (smartphones whose *CPU* has very limited resources or which do not have *GPU*). However, we remain convinced that this incompatibility is very temporary, thanks to the rapid changes in mobile hardware technology.

References

BATCHER, K. E. 1968. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, ACM, New York, NY, USA, AFIPS '68 (Spring), 307–314.

BETTIO, F., GOBBETTI, E., MARTON, F., AND PINTORE, G. 2007. High-quality networked terrain rendering from compressed bitstreams. In *Proceedings of the twelfth international conference on 3D web technology*, ACM, New York, NY, USA, Web3D '07, 37–44.

CELLIER, F., 2012. Openscalegl: http://openscales.org/news/openscalesgl-announce.html.

CIGNONI, P., GANOVELLI, F., GOBBETTI, E., MARTON, F., PONCHIO, F., AND SCOPIGNO, R. 2003. BDAM – batched dynamic adaptive meshes for high performance terrain visualization. *Computer Graphics Forum 22*, 3 (September), 505–514. Proc. Eurographics 2003.

CIGNONI, P., DI BENEDETTO, M., GANOVELLI, F., GOBBETTI, E., MARTON, F., AND SCOPIGNO, R., 2007. Ray-casted blockmaps for large urban models streaming and visualization, Sept.

DICK, C., KRUEGER, J., AND WESTERMANN, R. 2009. Gpu ray-casting for scalable terrain rendering. In *Proceedings of Eurographics 2009 - Areas Papers*, 43–50.

GARLAND, M., AND HECKBERT, P. S. 1997. Surface simplification using quadric error metrics. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, SIGGRAPH '97, 209–216.

GARLAND, M., AND SHAFFER, E. 2002. A multiphase approach to efficient surface simplification. In *Proceedings of the conference on Visualization '02*, IEEE Computer Society, Washington, DC, USA, VIS '02, 117–124.

GOBBETTI, E., MARTON, F., CIGNONI, P., DI BENEDETTO, M., AND GANOVELLI, F., 2006. C-bdam - compressed batched dynamic adaptive meshes for terrain rendering, sep. To appear in Eurographics 2006 conference proceedings.

GU, X., GORTLER, S. J., AND HOPPE, H. 2002. Geometry images. *ACM Trans. Graph. 21*, 3 (July), 355–361.

HOPPE, H. 1996. Progressive Meshes. ACM Press/ACM SIGGRAPH, New York, H. Rushmeier, Ed., 99–108.

HU, L., SANDER, P. V., AND HOPPE, H. 2009. Parallel view-dependent refinement of progressive meshes. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games*, ACM, New York, NY, USA, I3D '09, 169–176.

IGN, 2012. Géoportail: http://www.geoportail.fr.

JAMIN, C., GANDOIN, P.-M., AND AKKOUCHE, S. 2009. CHuMI Viewer: Compressive Huge Mesh Interactive Viewer. *Computer & Graphics 33*, 4 (Aug.).

LINDSTROM, P., AND TURK, G. 1998. Fast and memory efficient polygonal simplification. In *Proceedings of the conference on Visualization '98*, IEEE Computer Society Press, Los Alamitos, CA, USA, VIS '98, 279–286.

LOSASSO, F., AND HOPPE, H. 2004. Geometry clipmaps: terrain rendering using nested regular grids. *ACM Trans. Graph. 23*, 3 (Aug.), 769–776.

PAJAROLA, R., AND GOBBETTI, E. 2007. Survey of semi-regular multiresolution models for interactive terrain rendering. *Vis. Comput. 23*, 8 (July), 583–605.

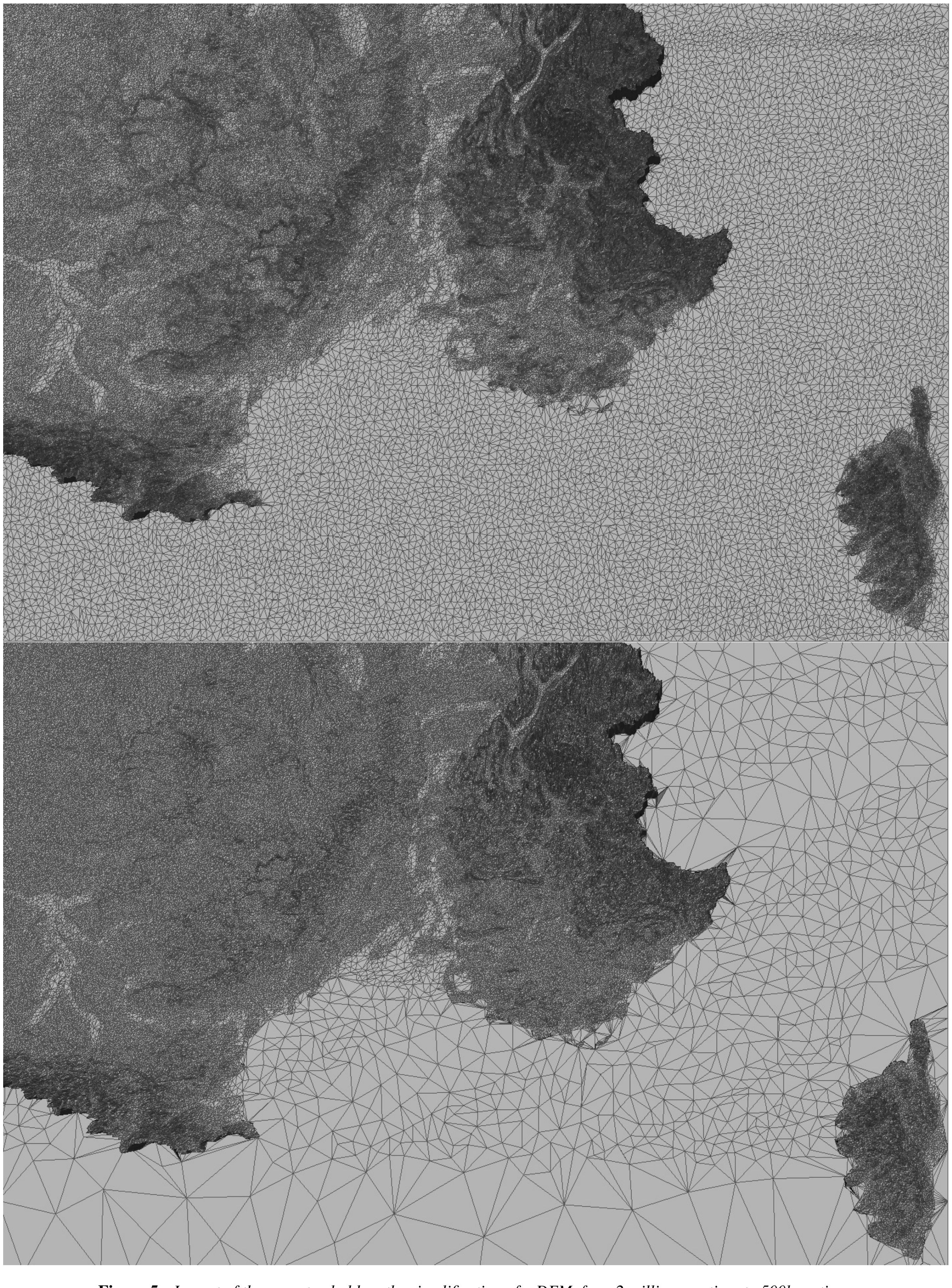

Figure 5: *Impact of the error treshold on the simplification of a DEM, from 2 millions vertices to 500k vertices.*