NON-UNIFORM MESH SIMPLIFICATION USING ADAPTATIVE MERGE PROCEDURES

Flávio Mello[1,2], Edilberto Strauss[2,3], Antônio Oliveira[2], and Aline Gesualdi[3]
[1]*Institute of Research and Development - IPD, Rio de Janeiro, Brazil;* [2]*Computer Graphics Laboratory - LCG - COPPE - UFRJ, Rio de Janeiro, Brazil;* [3]*Department of Electronics and Computer Engineering - DEL - POLI - UFRJ, Rio de Janeiro, Brazil*

Abstract: The performance of a walkthrough over terrain models is deeply influenced by the real scenario high level of details. To guarantee natural and smooth changes in a sequence of scenes, it is necessary to display the actual height field's maps at interactive frame rates. These frame rates can be accomplished by reducing the number of rendered geometric primitives without compromising the visual quality. This paper describes an optimized algorithm for building a triangular mesh, the terrain model, which combines an efficient regular grid representation with low cost memory requirements, using a bottom-up approach.

Key words: Terrain mesh simplification, quadtree, digital elevation model.

1. INTRODUCTION

The terrain walkthrough plays an important role in the virtual reality, as it is observed in computer systems like: Geographic Information Systems (GIS), Military Mission Planning, Flight Simulation, etc. A regular grid sampled data, known as Digital Elevation Model (DEM), is required (Turner, 2000) to represent terrain altimetry. However, the relationship between the actual map image resolution and its associated data can easily exceed the capabilities of typical graphics hardware, which makes impossible a real-time interactive application. A 3-dimensional clipping approach can be considered in order to reduce the geometric rendering primitives.

On the other hand, during a natural walkthrough, at each new viewpoint, the observer can better see the nearest portion of the map data set. So, scenes far from the viewer do not need to be rendered with a high level of details, such as those closer data. Thus, computer graphics methods associated to both view frustum culling and the view dependent continuous level of detail are useful to reduce rendering complexity.

In order to compute the DEM's triangular mesh, it is usually necessary that, at least, one of the following properties be fulfilled, as it is described in (Zhao, 2001; Lindstrom, 1996; Röttger, 1998; Hoppe, 1998; Blow, 2000a, 2000b; de Berg, 2000). A same terrain region triangulation may look different from each other according to the chosen properties. So, the mesh can:

- be *conforming*: when a triangle is not allowed to have a vertex of another triangle in the interior of one of its edges;
- *respect the input*: when the set of the resulting mesh vertex is included on the set of DEM's pixels coordinates;
- be *well shaped*: when the angles of any mesh triangle are neither be too large nor too small. Usually, it is required from the triangles angles to be in the range from 45^o to 90^o;
- be *non-uniform*: when it is fine near the borders of the components (pixels gradient values) and coarse far away from this borders.

On this paper we assume that the well shaped constrain can be relaxed. Also, it is considered that the vertices triangulation does not need to respect the height field components' edges. By this reason, it is allowed to add to the mesh extra points, called Steiner points (de Berg, 2000). The non-uniform mesh generation method we shall describe in this paper is based on a quadtree structure. Our algorithm divides the DEM into smaller triangular regions, and then merges the redundant triangles into bigger ones. It is conforming although it is neither well shaped nor respects the input.

2. RELATED WORK

In this section main algorithms on terrain rendering are briefly introduced. These algorithms attempt to represent surfaces with a given number of vertices, or within a given geometric error metric, or even trying to preserve application specific critical surface features.

Lindstrom et alii (1996) presented an algorithm for real-time level of detail reduction applied to high-complexity polygonal surface data. The algorithm uses a regular grid representation, and employs a variable screen-space threshold to bind the maximum error of the projected image. A coarse level of simplification is performed to select discrete levels of detail for

blocks of the surface mesh. Then, further simplification is done by performing a repolygonalization, where individual mesh vertices are considered for removal.

Duchaineauy et alii (1997) created the Real-time Optimally Adapting Meshes (ROAM). It is an evolution from Lindstrom's algorithm and it is much faster. It uses an explicit binary triangle tree structure. Although the ROAM is being widely used in games, Jonathan (2000a, 200b) reported a decline of performance for densely sample data.

Most of the work presented on this paper is based on Röttger's algorithm (Röttger, 1998). Röttger introduce a geomorphing algorithm, which operates top down on a quadtree data structure. The great advantage of Röttger's algorithm over Lindstrom's one is that it eliminates a phenomenon called vertex popping.

Hoppe (1998) describes a real time fly-over by using a View Dependant Progressive Meshes (VDPM) to terrains. He divides the terrain into several blocks, as the whole terrain is too large to fit in a single data structure. For each of the blocks, he generates triangular irregular networks according to viewpoints and error threshold. Though Hoppe's algorithm achieves a high frame rate, it demands too much storage and it lacks generality. Also, despite of producing a mesh with far less triangles than a regular grid based one, it spend too much time on optimization.

3. OVERVIEW

The underlying data structure of the presented algorithm is basically a quadtree. For the discussion in this paper, it is assumed that the height fields dimensions are $2^n x 2^m$, where n and m might not be equal. The mesh generation presented here is will be described as a sequence of two steps.

First, the height field is recursively divided into four quadrants. Every tree node is divided until a predefined and customized tree height is reached. Since no simplification criteria over the height field points are made, the resulting tree is a full divided quadtree. Every node on the tree corresponds to a square patch (triangle pair) of the height field. This implies that squares represented by tree leaves correspond to the most refined subdivision of the height field, called quadtree subdivision. This subdivision represents a regular and uniform grid of the DEM.

The second step of the algorithm implements the merge of redundant triangles into bigger ones. It is performed a preorder tree walk, where only leaves are inserted into a triangle pair list (TPL). The merge of the redundant triangles occurs during the insertion into the TPL. At the end of the tree leaves insertion procedure, different sizes triangle pairs will compose the

TPL, which represents the optimized mesh. The TPL is a double linked ordered list, where its triangle pairs are order first by their horizontal dimension, and then, by their vertical dimension.

During the insertion procedure, each leaf node may merge with a TPL node if they have two properties. First, nodes must have a coincident edge, which means that its edges should not only be adjacent, but also have the same side sizes. At first, it is checked if horizontal merges may occur, and then, if vertical direction merges may occur. Figure 1 illustrates this property by pointing out two patches performing a horizontal merge, and two patches not concluding their merge.

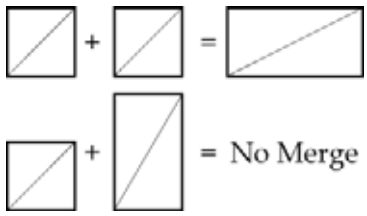

Figure 1. Two neighbor squares may merge if their coincident edges are the same size.

The second merge property sets that nodes must have the same topology. It demands that the squares represented by nodes must be at the same spatial plane, or with the same normal, as shown in Figure 2. So, nodes merge may occur only when they are coplanar patches with same size coincident edges.

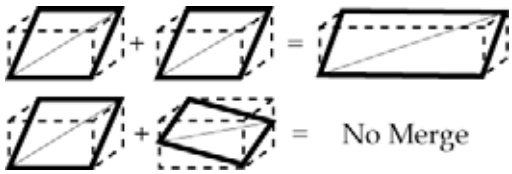

Figure 2. Merge may occur only with coplanar patches.

It should be observed that the merge of two nodes might result into a new larger node. This new node will also be tested against the others TPL elements, either on horizontal, or on vertical direction. So, many merges might occur by inserting just one quadtree leaf node into a non-empty TPL.

If it is not possible to perform a merge between the tree leaf node and an element from the TPL, then the quadtree node is inserted in the list first position. However, if the merge criteria are satisfied, then the list element is removed from the TPL and two nodes became a new one. It is important to observe that this new node may also be combined with other TPL elements,

either on the horizontal, or on the vertical direction. Thus, a node insertion can lead to many triangle pair merges.

The merging behavior varies according to the component (pixel value gradient) position over the quadtree subdivision. Consider a node where the next step of the quadtree subdivision is composed by four leaves from the tree. The height component that pulls a vertex to a higher or lower altitude may be located at one of the three positions illustrated on Figure 3a. The component may be coincident to vertex neighboring all four patches (patches 5-6-7-8); or coincident to the vertex neighboring just two patches (patches 5-7), or even coincident to the corner of the quadtree subdivision (patch 7). These configurations for the height component implies on three primitive patterns of patches, created by the proposed algorithm, as shown on Figure 3b. The Figure 3c represents the quadtree subdivision by Röttger (1998) and Lindstrom (1996) just before determining how should be the triangle fan configured in order to avoid the cracks. On the first and second cases, the proposed method used fewer triangles than Röttger and Lindstrom even before their attempt to eliminate the cracks. On the third case the method uses more rendering triangles than the other methods. Since the other methods still need to do some splitting before drawing the triangulated height field patch, it is expected that the presented algorithm will need less triangles to render the patch on this case too.

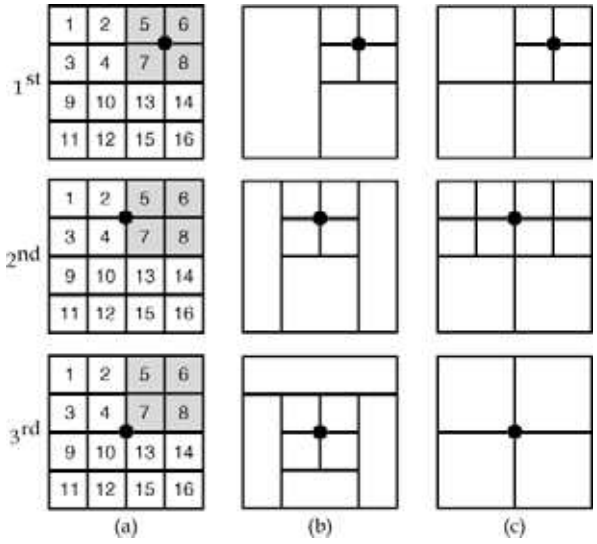

Figure 3. Different DEM subdivision according to the height component position, for components in the interior of the height map. (a) Original Subdivision; (b) Adaptative Merge Subdivision; (c) Röttger Subdivision.

The probability p_1 of occurring a height component into the interior of the first case patch is given by 4/9. In other words, for the first case, there are 4 possible component height positions against 9 available positions. As it can be observed, the proposed method represents the first case patch using 6 triangles, while Röttger method uses 7. So, the rendering triangles drawing rate (RDTR) between the proposed algorithm and Röttger one, on this case, is given by 6/7. Therefore, the total rendering triangles drawing rate when a height component occurs into the interior of a patch is given by

$$RTDR_{\text{interior}} = \sum_{case=1}^{3} p_{case} \cdot RTDR_{case} = \left(\frac{4}{9} \cdot \frac{6}{7}\right) + \left(\frac{4}{9} \cdot \frac{7}{10}\right) + \left(\frac{1}{9} \cdot \frac{8}{4}\right) = 0.9143 \quad (1)$$

The value obtained by equation 1 indicates that the proposed algorithm would used 8.57% less triangles than Röttger method.

A similar analysis can be made on the border of the height field. The height component can be located at one of the following position from Figure 4a. The component may be coincident to vertex neighboring two patches of two different nodes (patches 3-9); or coincident to the vertex neighboring just two patches of a same node (patches 1-3), or even coincident to the corner of the height field (patch 1). These configurations

for the height component implies on three border primitive patterns of patches, created by the proposed algorithm, as shown on Figure 4b. Again, the Figure 4c represents the quadtree subdivision by Röttger, just before determining how should be the triangle fan configured in order to avoid the cracks.

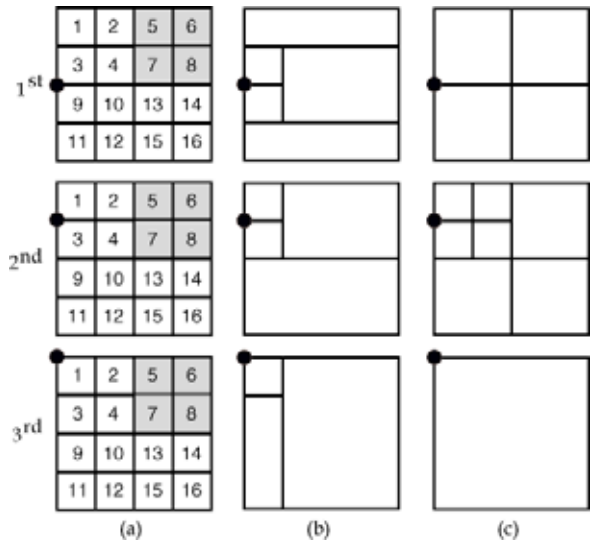

Figure 4. Different DEM subdivision according to the height component border position, for components at the border of the height map. (a) Original Subdivision; (b) Adaptative Merge Subdivision; (c) Röttger Subdivision.

The $RDTR_{border}$ of the height field may be computed on a similar way to the $RDTR_{interior}$ calculus. The probability p_1 of occurring a height component into the interior of the first case patch is given by 4/16. So, for the first case from Figure 4, there are 4 possible component height positions against 16 available positions. The proposed method represents the first case patch using 5 triangles, while Röttger method uses 4. So, the rendering triangles drawing rate (RDTR) between the proposed algorithm and Röttger one, on this case, is given by 5/4. Note that by computing the $RDTR_{border}$, it would reach the value of 1.3482 (see equation 2). This means that the proposed algorithm needs 34.82% more triangles than Röttger method, for occurrences of components on the height map borders.

$$RTDR_{border} = \sum_{case=1}^{3} p_{case} \cdot RTDR_{case} = \left(\frac{4}{16} \cdot \frac{5}{4}\right) + \left(\frac{8}{16} \cdot \frac{4}{7}\right) + \left(\frac{4}{16} \cdot \frac{3}{1}\right) = 1.3482 \quad (2)$$

Although Röttger algorithm has greater performance at the border of the height field, the proposed algorithm is still more efficient due to the gain obtained inside of the height field. It must be observed that the number of inner triangles grows much faster than the number of border triangles, as the quadtree height increases. In fact, the algorithm gets better result than Röttger's one when the quadtree height increases from 4 to 5. Real applications frequently uses quadtree height values between 7 and 9 (Ögren , 2000).

4. ADAPTATIVE MERGE ALGORITHM' COMPLEXITY ANALYSIS

The Adaptative Merge Algorithm is composed by three stages: (1) the construction of the quadtree with a user predefined value d for the quadtree height; (2) the construction of an intermediate triangle pair list in order to avoid recursive procedures; (3) the nodes removal from the intermediate list and its insertion into the TPL. Once these stages describe a sequential algorithm, the mean time of the whole procedure is defined by the mean time sum of each stage

$$T(n) = T_1(n) + T_2(n) + T_3(n) \tag{3}$$

where the term $T_1(n)$ represents the necessary time to create all nodes from the full divided tree with height d, $T_2(n)$ is the time to construct the intermediate list, and $T_3(n)$ is the time for constructing the TPL.

On a full divided quadtree, the first tree level has just 1 node, the second level has 4 nodes, the third level has 16 nodes, and so on. So, the sum S of all nodes created until the user predefined quadtree height (d+1 level) is reached can be represented by an equation of the form:

$$S = 4^0 + 4^1 + 4^2 + \cdots + 4^d = \frac{1\left(4^{d+1} - 1\right)}{4 - 1} \tag{4}$$

At the end of the quadtree construction, it is desired to obtain n leaves nodes, which means, 4n-1 nodes into the whole tree. So, the substitution 4^d=n, on the equation 4, gives us:

$$S = \frac{1\left(4 \times 4^d - 1\right)}{4 - 1} = \frac{1\left(4 \times n - 1\right)}{4 - 1} = \frac{4n - 1}{3} \tag{5}$$

Therefore, according to the equation 5, the time spent to construct the full divided quad tree is limited to:

$$T_1(n) = \theta\left(\frac{4n-1}{3}\right) = \theta(n) \tag{6}$$

The next stage, associated to the construction of the intermediate triangle pair list. On this stage it is performed a preorder tree walk where just the n leaves nodes are inserted into the first position of an intermediate linked list. This tree walkthrough is executed as many times as are the tree elements.

It is considered that each tree node can be visited within a constant time complexity. As the tree leaf nodes are inserted at the intermediate linked list head. It is reasonable to consider that the insertion on this list can also be computed within a constant time complexity. Using equation 5 it is possible to conclude that the total amount of preorder visited nodes is given by (4n-1)/3. Similarly to $T_1(n)$ time, the $T_2(n)$ time spent on this stage is $\theta(n)$, as it is widely known on the literature (Markenzon, 1994; Cormen, 1997). Therefore, the equation 3 can be rewritten as:

$$T(n) = \theta(n) + \theta(n) + T_3(n) \tag{7}$$

The TPL containing the triangle pairs, is represented by a chaining hash table L, with a hash function h, as it is described on Figure 5. The number of slots from L is defined by m.

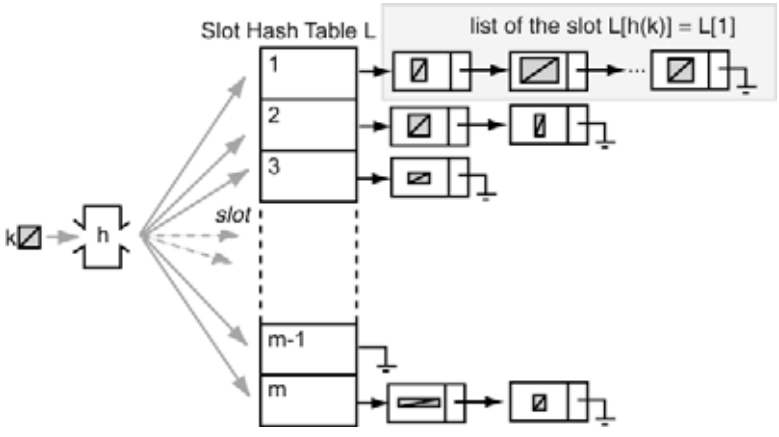

Figure 5. The chaining hash table schema for the Triangle Pair List (TPL).

The worst case analysis for a chaining hash table is given when all n entries from the L table are mapped into the same slot, creating a linked list of size n (Cormen, 1997). Fortunately, this is an impossible case for the adaptative merge algorithm because the hash table slots represent uniform distributed regions of a regular grid placed over the height map. No region overlap is allowed. So, the hash table loading factor is defined by α=n/m, which represents the average list size of the chaining hash table (Cormen, 1997).

For an entry k, that is, a triangle pair k, it is considered that the hashing value h(k) might be algebraically computed in O(1). So, once computed the hash table slot where the triangle pair must be inserted, it is necessary to find out if it is possible to merge the triangle pair k with an L[h(k)] element. The time spent to fetch an element in order to merge it with the entry k depends linearly on the size of the list associated to the slot L[h(k)]. There are two fetch cases that must be considered: (1) an unsuccessful fetch, where there are no possible merges with k; (2) a successful fetch, where there is a list element that might merge with k.

Consider the first case. The hashing value h(k) can be computed in O(1), and the entry k can be placed into any of the m cells list. As a result, the average time for an unsuccessful search is the time spent to get in the end of one of the m slots' linked list. As the average size of a chained hash table linked list is defined by its loading factor α=n/m, it is possible to assert that the average time for an unsuccessful search, including the h(k) calculus, is given by:

$$t_f(k) = \theta(1+\alpha) = \theta(\alpha) \tag{8}$$

Then, consider a successful search. The linked list chained hash table element might be on the i^{th} list position of one of the m table's slot. In order to search for the element i, it is necessary walk through i list elements. So the average time of a successful search is given by:

$$\frac{1}{n}\sum_{i=1}^{n}\frac{i}{m} = \frac{1}{nm}\sum_{i=1}^{n} i = \frac{1}{nm}\frac{(n+1)n}{2} = \frac{n}{2m} + \frac{1}{2m} = \frac{\alpha}{2m} + \frac{1}{2m} \tag{9}$$

The mean time for a successful search t_s, including the h(k) calculus, is given by:

$$t_s(k) = \theta\left(1 + \frac{\alpha}{2} + \frac{1}{2m}\right) = \theta(\alpha) \tag{10}$$

Consequently, at the worst case, the equations 8 and 10 point out that the average time for searching operation, inside of one of the m list from the L hash table, is not dependant on a successful or unsuccessful search. Both $t_f(k)$ and $t_s(k)$ depends on α, i.e., it is conditioned to the size of the list. This is a well-known result that can be better studied on Markenzon (1994) and Cormen (1997).

The time complexity of just one chained hash table insertion $T_3(n)/n$ is represented by the form:

$$\frac{T_3(n)}{n} = p_f t_f(k) + p_s t_{s_\alpha}(k) = \frac{1}{\alpha+1} t_f(k) + \frac{\alpha}{\alpha+1} t_{s_\alpha}(k) \qquad (11)$$

where: p_f is the probability of not finding any node to merge with k during the insertion on the α sized list; p_s is the probability of finding a node that can perform a merge with k; t_f is the time for an unsuccessful search of an element; and t_s if the time for a successful search.

On the previous section, it was defined that the merge of a list element with a triangle pair k might occur only when the polygons are coplanar with coincident edges. By the time this condition is satisfied, the chained hash table linked list element is removed from the data structure, and then the two triangle pairs are merged, resulting on a new triangle pair k` with bigger dimensions. This new triangle pair k`, during its insertion on the list, might also be merged with another chained hash table linked list element.

Thus, just one merging operation can result into several others merging operations. In fact, when a successful search is detected, it is necessary to remove the element from the α sized list and merge it with the k entry. This procedure is recursively activated, gradually reducing the list size until an unsuccessful search is reached. Hence, $t_{s\,\alpha}$ can be rewritten as:

$$\begin{aligned} t_{s_\alpha}(k) &= p_{f_{\alpha-1}} t_f(k) + p_{s_{\alpha-1}} t_{s_{\alpha-1}}(k) \\ &= \frac{1}{(\alpha-1)+1} t_f(k) + \frac{\alpha-1}{(\alpha-1)+1} t_{s_{\alpha-1}}(k) \\ &= \frac{1}{\alpha} t_f(k) + \frac{\alpha-1}{\alpha} t_{s_{\alpha-1}} \end{aligned} \qquad (12)$$

Therefore, considering the worst case, the list might indefinitely be operated (insertion, removal and merge) until it becomes empty. The time progression $t_{s\,i}(k)$ associated to the successful searches into the table L might be described as follows:

$$
\begin{aligned}
t_{s_{\alpha-1}}(k) &= p_{f_{\alpha-2}} t_f(k) + p_{s_{\alpha-2}} t_{s_{\alpha-2}}(k) \\
&= \frac{1}{(\alpha-2)+1} t_f(k) + \frac{\alpha-2}{(\alpha-2)+1} t_{s_{\alpha-2}}(k) \\
&= \frac{1}{\alpha-1} t_f(k) + \frac{\alpha-2}{\alpha-1} t_{s_{\alpha-2}}(k) \\
\vdots \quad &\quad \vdots \qquad\qquad \vdots \\
t_{s_3}(k) &= p_{f_2} t_f(k) + p_{s_2} t_{s_2}(k) \\
&= \frac{1}{2+1} t_f(k) + \frac{2}{2+1} t_{s_2}(k) \\
&= \frac{1}{3} t_f(k) + \frac{2}{3} t_{s_2}(k) \\
t_{s_2}(k) &= p_{f_1} t_f(k) + p_{s_1} t_{s_1}(k) \\
&= \frac{1}{1+1} t_f(k) + \frac{1}{1+1} t_{s_1}(k) \\
&= \frac{1}{2} t_f(k) + \frac{1}{2} t_{s_1}(k) \\
t_{s_1}(k) &= p_{f_0} t_f(k) + p_{s_0} t_{s_0}(k) \\
&= \frac{1}{1} \times 0 + 0 \times 0 \\
&= 0
\end{aligned}
\qquad (13)
$$

Now, recompiling the values $t_{s\,i}(k)$ according to the results obtained on equation 13 we have:

$$
\begin{aligned}
t_{s_2}(k) &= \frac{1}{2}\cdot 1+\frac{1}{2}\cdot 0 \\
&= \frac{1}{2} \\
t_{s_3}(k) &= \frac{1}{3}\cdot 2+\frac{2}{3}\cdot\frac{1}{2} \\
&= \frac{2}{3}+\frac{2}{3}\frac{1}{2} \\
t_{s_4}(k) &= \frac{1}{4}\cdot 3+\frac{3}{4}\left(\frac{2}{3}+\frac{2}{3}\frac{1}{2}\right) \\
&= \frac{2}{3}+\frac{3}{4}\frac{2}{3}+\frac{3}{4}\frac{2}{3}\frac{1}{2} \\
t_{s_5}(k) &= \frac{1}{5}\cdot 4+\frac{4}{5}\left(\frac{3}{4}+\frac{3}{4}\frac{2}{3}+\frac{3}{4}\frac{2}{3}\frac{1}{2}\right) \\
&= \frac{4}{5}+\frac{4}{5}\frac{3}{4}+\frac{4}{5}\frac{3}{4}\frac{2}{3}+\frac{4}{5}\frac{3}{4}\frac{2}{3}\frac{1}{2} \\
\vdots \quad \vdots & \qquad\qquad \vdots
\end{aligned}
\tag{14}
$$

Performing the appropriated substitution of equation 14 until the iteration from the equation 11, we obtain:

$$\begin{aligned}
\frac{T_3(n)}{n} &= \left(\frac{\alpha}{\alpha+1}\right)+ \\
&\quad \left(\frac{\alpha}{\alpha+1}\frac{\alpha-1}{\alpha}\right)+ \\
&\quad \left(\frac{\alpha}{\alpha+1}\frac{\alpha-1}{\alpha}\frac{\alpha-2}{\alpha-1}\right)+ \\
&\quad \left(\frac{\alpha}{\alpha+1}\frac{\alpha-1}{\alpha}\frac{\alpha-2}{\alpha-1}\frac{\alpha-3}{\alpha-2}\right)+\cdots+ \\
&\quad \left(\frac{\alpha}{\alpha+1}\frac{\alpha-1}{\alpha}\frac{\alpha-2}{\alpha-1}\frac{\alpha-3}{\alpha-2}\cdots\frac{4}{5}\frac{3}{4}\frac{2}{3}\frac{1}{2}\right) \\
&= \frac{\alpha}{\alpha+1}+\frac{\alpha-1}{\alpha+1}+\frac{\alpha-2}{\alpha+1}+\frac{\alpha-3}{\alpha+1}+\cdots+\frac{1}{\alpha+1} \\
&= \frac{1}{\alpha+1}\sum_{i=1}^{\alpha} i \\
&= \frac{1}{\alpha+1}\frac{(\alpha+1)\alpha}{2} = \alpha/2
\end{aligned} \tag{15}$$

So, according to equation 15 the time for constructing the whole triangle pair list is given by:

$$T_3(n) = \frac{n\alpha}{2} = \frac{n}{2}\frac{n}{m} = \frac{n^2}{2m} \Rightarrow T_3(n) = \theta(n^2) \tag{16}$$

Finally, through equation 7 it is possible to affirm that the average time for computing the TPL (not for rendering it) is given by:

$$T(n) = \theta(n) + \theta(n) + \theta(n^2) = \theta(n^2) \tag{17}$$

5. RESULTS

All screen shots have been taken from the application running on an ordinary PC with a 1GHz processor and a GeForce 2 MX 400 video board.

The 1024x1024 grid of Figure 6a represents actual elevation data of Salt Lake City West, taken from the United States Geological Survey (USGS).

The elevation at each grid vertex is given as an integer in the range 0-255, where one unit represents 8.125 meters. Figure 6b represents a 2048x2048 grid extracted from Rio de Janeiro City height field. It was taken from Instituto Militar de Engenharia (IME) and represents Urca, Botafogo, Leme and Copacabana neighborhoods. One unit elevation from this grid vertex represents 0.418 meters.

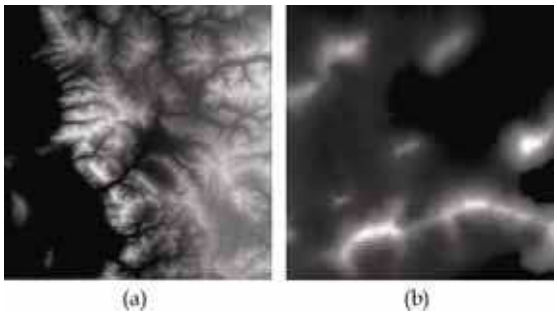

Figure 6. Height field used for examples: (a) Salt Lake City West 1024x1024 grid; (b) Rio de Janeiro City Neighborhoods 2048x2048 grid.

A regular grid, from Figure 6a region, generated by a quadtree subdivision would take 524.288 triangles, since the quadtree height was set to 9. By applying the merge techniques described on this paper, the proposed method generates a rendering list with 384.146 triangles, while Röttger method (Röttger, 2004) takes 461.104 triangles. It represents a 26,73% of optimization when compared to the regular grid, and 16,69% of optimization when compared to Röttger method (Röttger, 2004).

Considering the Figure 6b case, the regular grid generated by a quadtree subdivision would also take 524.288 triangles, because the quadtree height is the same for both examples. The proposed method generates a rendering list with just 313.052 triangles, while Röttger method (Röttger, 2004) takes 378.036 triangles. It represents a 40,29% of optimization when compared to the regular grid, and 17,19% of optimization when compared to Röttger method (Röttger, 2004).

The region presented on Figure 7 corresponds to the Salt Lake City mesh created by the proposed algorithm, while Figure 8 corresponds to Rio de Janeiro neighborhoods mesh. At these examples, the quadtree height was set to 6 for both regions because values greater than this one produce too overcrowded figures.

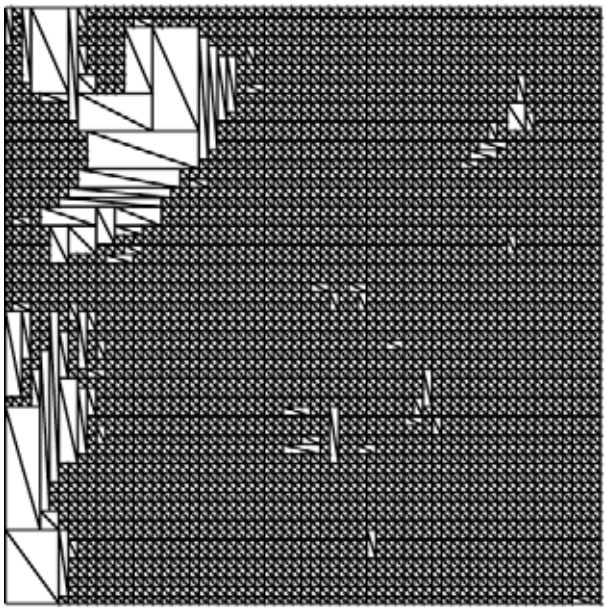

Figure 7. Top view from Salt Lake City West generated mesh.

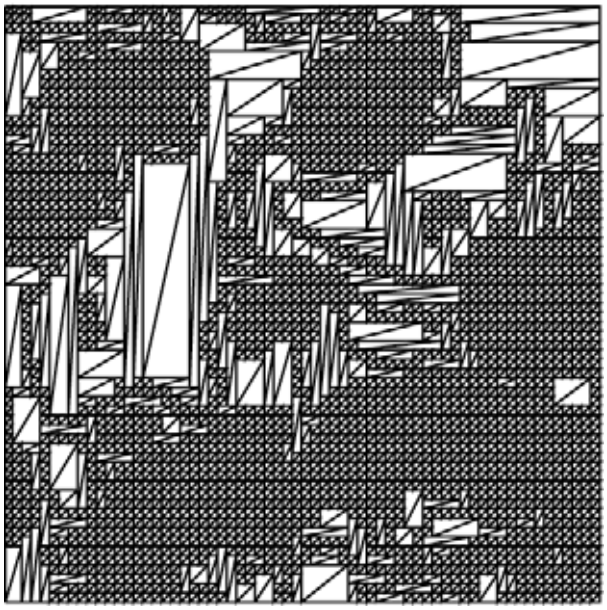

Figure 8. Top view from Rio de Janeiro City Neighborhoods generated mesh.

Finally, it is illustrated two typical valley views from each DEM sample. Figure 9 shows the Salt Lake City West mesh covered by a pseudo-color texture image, using a syntetic sky for day light representation. The Figure 10 shows Rio de Janeiro mesh texturized with an Ikonos satellite image, provided by Instituto Militar de Engenharia.

Figure 9. Snapshot from a Salt Lake City West valley view. The mesh is covered by a pseudo-color texture image and the day light sky was synthetically generated.

Figure 10. Snapshot from a Rio de Janeiro City Neighborhoods valley view. The mesh is covered by an Ikonos satellite image and the night sky was synthetically generated.

6. CONCLUSION

It was presented a bottom-up algorithm for optimizing terrain mesh triangulations. The method has been implemented and provides high quality

triangulations with thousands of geometric primitives. The generated mesh is conforming, although it is neither well shaped nor respects the input. The coherence between frames has not been exploited yet, but it has achieved good frame rates on PC platforms, such as 47 fps. Critical future issues include level of detail rendering and efficient paging mechanism, which will allow rendering height fields that do not entirely fit into RAM.

REFERENCES

Blow, Jonathan. (2000). *Terrain Rendering at High Levels of Detail*, Paper for the Game Developers' Conference 2000, San Jose, California, USA.

Blow, Jonathan. (2000). *Terrain Rendering Research for Games*, Slides for Siggraph 2000 Course 39.

Cormen, Thomas H., Charles Leiserson, Ronald Rivest. (1997). *Introduction to Algorithms*, MIT Press, 18ed., p.221-243.

de Berg, Mark, et alii. (2000). *Computional Geometry - Algorithms and Applications*, 2ed., Berlin, Springer, c.14.

Duchaineauy, Mark, Murray Wolinsky, et alii. (1997). *ROAMing Terrain: Real-time Optimally Adapting Meshes*, IEEE Visualization '97 Proceedings.

Hoppe, H. (1998). *Smooth View Dependant Level-of-Detail Control and its Application to Terrain Rendering*, Technical Report, Microsoft Research.

Lindstrom, P., D. Koller, et alli. (1996). *Real-time continuous level of detail rendering of height fields*, Computer Graphics, SIGGRAPH '96 Proceedings, p.109-118.

Markenzon, Lilian, Jayme Luiz Szwarcfiter. (1994). *Data Structures and its Algorithms*, Livros Técnicos e Científicos. (in portuguese)

Ögren, A. (2000). *Continuous Level od Detail in Real-Time Terrain Rendering*, MSc. Dissertation, University of Umea, January, 2000.

Röttger, S., W. Heidrich, P. Slasallek and H. Seidel. (1998). *Real-Time Generation of Continuous Levels of Detail for Height Fields*, WSCG '98 Proceedings, p. 315-322.

Röttger, S. (2004). *Terrain LOD Implementations - libMini*, http://www.vterrain.org/LOD/Implementations/. [capture on 26/03/04]

Turner, Bryan. (2000). *Real-Time Dynamic Level of Detail Terrain Rendering with ROAM*. (www.gamasutra.com/features/20000403/turner_01.htm)

Zhao, Youbing, Ji Zhou, Jiaoying Shi and Zhigeng Pan. (2001). *A Fast Algorithm For Large Scale Terrain Walkthrough*, CAD/Graphics.