# A Real-Time Single-Pass Continuous LOD Algorithm for Regular Height Maps

CESAR TADEU POZZER[1]
MARCELO DREUX[2]
BRUNO FEIJÓ[1]

[1]ICAD/IGAMES - PUC-Rio, Brazil
[1]Computer Science Department – PUC-Rio, Brazil
{pozzer,bruno}@inf.puc-rio.br

[2]Mechanical Engineering Department – PUC-Rio, Brazil
dreux@mec.puc-rio.br

**Abstract**
*This paper presents an efficient LOD algorithm that selects the appropriate resolution and renders the resulted irregular mesh in a single pass through a quadtree. In order to select the appropriate resolution it makes use of quadtree neighboring resolution, surface geometry and viewer position.*
**Keywords:** *LOD, terrain, height map, quadtree*

## 1 Introduction

Level of Detail (LOD) schemes have crucial importance in speedup real-time graphical applications that handle large terrains or objects with a large number of faces. For instance, in virtual environments, Geographic Information Systems (GIS) and real-time outdoor games, like flight simulators and massively multiplayer games. The basic idea is to reduce the amount of triangles to be pushed through the rendering pipeline, by means of polygonal surface approximation, without compromising visual quality.

Such terrains are usually represented as regular height fields. Therefore schemes for adaptive refinement based on both viewer position and local terrain geometry can be used to produce irregular meshes with fewer triangles, in order to increase the overall performance of the system. Intuitively, planar regions represented by larger triangles, while non-planar by smaller ones.

When developing a LOD algorithm for height fields, one of the most challenging tasks it to guarantee smooth continuity within the blocks boundary, in order to avoid T-junctions. This is especially true when terrain roughness is considered in the process. Approaches that consider just viewer distance tend to be easier to implement. However, they may hide some visual details and, with viewpoint moving, the triangulation is continuously changing, resulting in a phenomenon called “vertex popping”. As the observer approaches an area with detailed information, the detail will appear all of a sudden [1].

The paper proposes a new LOD algorithm that is applicable to surfaces that are represented as uniformly-gridded height fields. Beyond this, it features the following characteristics:

- **Easy implementation**: Typically, most LOD algorithms are very complex to implement and comprehend, and thus they are difficult to be adapted to graphical applications. The goal is to create a simplified and efficient framework that implements a LOD algorithm and allows game programmers, with few effort, to adapt the code and to visualize terrains

that can emerge directly from 8/16-bit monochromatic raw images;

- **No pre-processing steps**: Some LOD algorithms' performance is based on some large pre-processed data set. This data set can be generated just before the visualization starts [2], what causes a considerable delay, or stored in pre-computed files, what requires additional algorithms and some effort from the user. As an example, Ulrich [3] presents an efficient, smooth vertex morphing that avoids vertex popping, although, non-trivial preprocessing is required and the data set must be static. As another example, Corpes [4] makes heavy use of pre-computed vertex buffers, for each possible LOD tile. Even without some pre-processing steps, the proposed algorithm presents fast performance, in both rendering and quadtree initialization;
- **Smoothness between different surface levels of detail:** Even though it does not work with geomorphing [1, 9], the algorithm takes into account both viewer position and local terrain geometry, in order to smoothly change the surface geometry between consecutive frames. In this process, it also avoids T-junctions. In a recent work, Losasso [5] simplifies meshes purely on distance, so that LOD is independent of the data content. It is assumed that practical terrains are well behaved, and usually do not have high-contrasts, a feature that is not realistically dealt with by that algorithm;
- **One-pass algorithm**: Certainly, the main contribution of the present work is the rendering and mesh simplification which are integrated into a single pass. Hence, it achieves a better efficiency, since quadtree traversal is performed once a frame. It has been designed a data structure specifically for this purpose. Many approaches such as [2, 6], use two passes algorithms: one for refinement (update) and another for rendering.

The following section of this paper describes the theory and procedures necessary for implementing the proposed algorithm. In section 3 the performance of the proposed algorithm with the work of Ulrich [2], from which many programming tricks were borrowed. This session also presents limitations of the current algorithm related to the refinement speed. To finalize, conclusions and future work are presented.

## 2 Mesh Refinement and Rendering

### 2.1 Data Structure

For better understanding of the proposed algorithm, the underlying quadtree data structure is introduced. It encodes height data of size $2^n+1$ x $2^n+1$ at different resolutions in different levels of hierarchy. For rendering purposes, each quadrant is divided, by two diagonals, in four regions (*left*, *up*, *right* and *bottom*), as shown in Figure 1b, which can be individually triangulated in a clock-wise order according to appropriate local resolution. Each quadrant only stores the height of the vertices corners in 16-bit integer format.

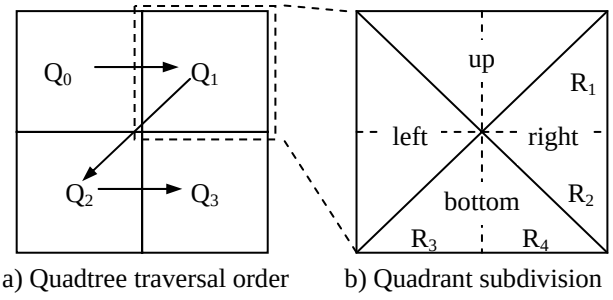



Figure 1: (a) Quadtree data structure. (b) Dashed lines represent edges that may be enabled. $R_n$s store the local mesh resolution

Neighboring resolutions are only stored for the *right* and *bottom* regions, since quadtree is traversed from the top left most corner to the bottom right most corner (Figure 1a). More specifically, the *right* region is associated with resolutions $R_1$ and $R_2$, and the *bottom* with $R_3$ and $R_4$, as shown in Figure 1b. Figure 2 presents a complete example of all required neighboring resolution, necessary to determine

the possible resolutions at each quadrant regions. As can be observed in this figure, the shaded quadrant must have pointers to four others, which are the ones that may influence the local resolution. Arrows indicate resolutions that should be queried in order to avoid T-junctions among neighboring quadrants rendered at different resolutions. Queries only occur among quadrants of the same level in the hierarchy. For example, for rendering the *right* quadrant, it is necessary to check $R_2$ from the *upper* quadrant and $R_3$ from the *upper-right diagonal* quadrant, which are represented as dashed triangular regions. The general rule is that the difference between the local $R_L$ and target $R_T$ resolutions cannot be greater that one. Additional pointers for each one of the four children are also stored in each node of the quadtree.

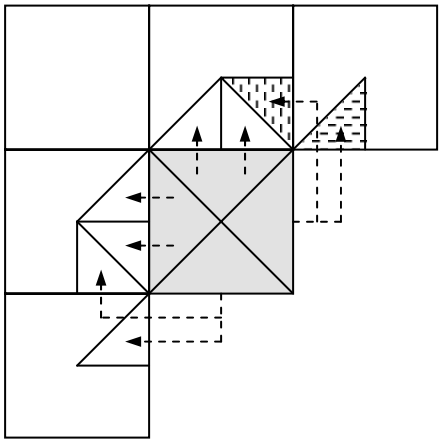


Figure 2: Required neighboring information for rendering a quadrant

When rendering a quadrant, not necessarily all neighbors are valid, and in this case they are simply discarded. The first rendered quadrant is the best example, because none of the neighbors exist (Figure 3). When building the quadtree, such quadrants can be detected and all invalid pointers are forced to point themselves.

### 2.2 Traversal Algorithm

Figure 4 shows a diagram of the traversal algorithm: 1) selects the appropriate rendering level *l* of each quadrant, 2) defines the current resolution for each region (by enabling or disabling some edges), and 3) renders the irregular mesh of triangles, with just a single traversal through the quadtree nodes. It uses a common recursive approach that traverses the quadtree in a top-down fashion, starting from the root node.

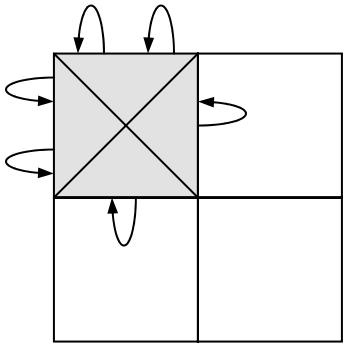


Figure 3: First quadrant neighbors

The recursion stops one level before the leaves of the tree. During the traversal, a node can be rendered if its depth fulfills some constraints:

$$(\min(lr, ur) - 1) \leq depth \leq (1 + \max(lr, ur)),$$

where *lr* and *ur* are the *left* and *up* quadrants resolution.

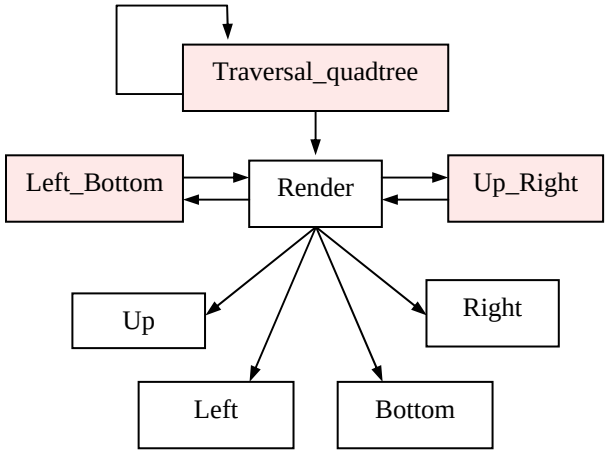



Figure 4: Traversal algorithm. Each block represents a function. The recursive ones are highlighted

As nodes are being visited by the general *Render* function, if they have the desired resolution level, the associated vertices are rendered. If not, the depth-first recursive search continues for that branch. There are six special rendering functions: *up*, *left*, *bottom*, *right*, *left_bottom* and *up_right*. They are called accordingly to some patterns presented in

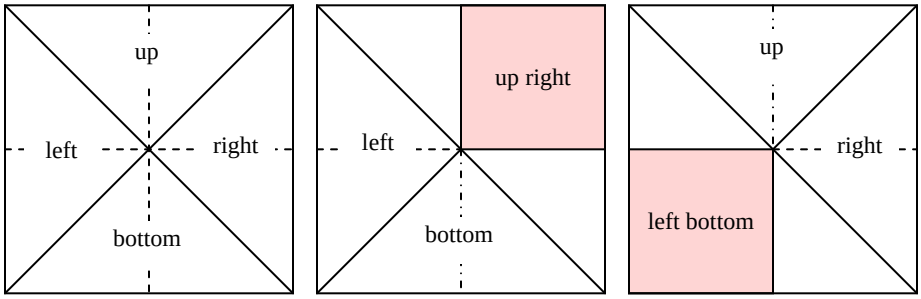



Figure 5: Different patterns for rendering/refining a quadrant

Figure 5, which are originated from all possible neighboring resolutions a quadrant may have. The first one is the only non-recursive pattern, and can be triangulated with minimum of 4 and maximum of 8 triangles. The other patterns are rendered with recursive and non-recursive functions, as described below. Similarly to [2, 6], the smallest mesh that uses this triangulation has dimension 3x3 vertices.

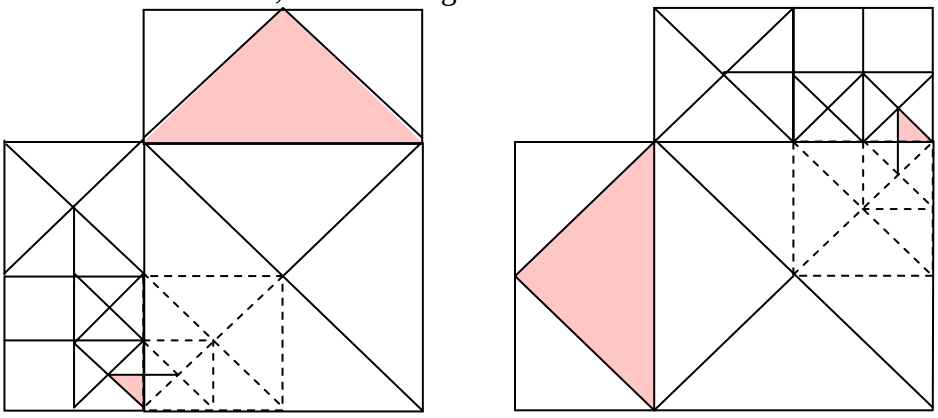
The recursive patterns should be used when $R_3$ and $R_4$, from the *up* quadrant, or $R_1$ and $R_2$, from the *left* quadrant, do not have the same resolution. For the examples presented in Figure 6, the difference in resolution is 3, but it could be greater, what would result in more recursive steps. Both recursive patterns finish the recursive rendering with the non-recursive one.

The resolution of a node is only defined when the traversal reaches a quadrant that can represent the appropriate local resolution, by considering neighboring resolution constraints, terrain geometry and viewer position. This node is considered a terminal node, and is assigned with the resolutions of the *right* and *bottom* regions, as presented in Figure 1.

When the algorithm traverses the quadtree towards the leaves in this recursive construction of the mesh, it continuously checks neighboring resolutions. As pointed out in Session 3.1, queries for neighboring resolutions only occurs among quadrants of the same level. To allow such queries, all parents are assigned with the highest resolution of their three children, as represented by Figure 7. It happens when the depth search returns to higher nodes. This ensures that all $R_n$, in all higher nodes, contain the real resolution the quadrant was rendered at lower levels.

### 2.3 Simplification criteria

As pointed out in the previous session, terrain geometry is also considered when selecting a quadrant resolution. A "vertex interpolation error" [2] is used, which consists in evaluating the difference in height between



Left_bottom recursion  Up_right recursion

Figure 6: Examples of recursive patterns (dashed) necessary to match different resolutions on a quadrant neighboring (shaded).

the correct location of a vertex, and the height of the edge in the triangle which approximates the vertex when it is disabled (Figure 8).

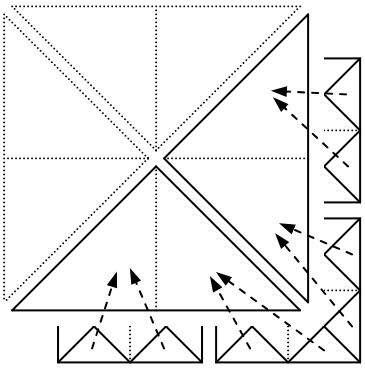


Figure 7: Back propagation of children resolutions

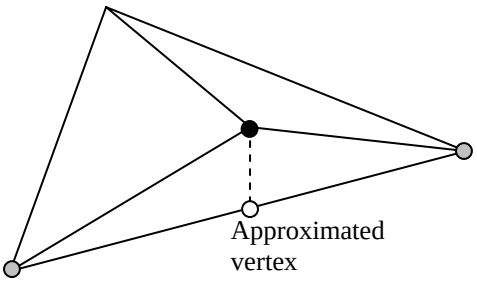



Figure 8: Vertex interpolation error [2]

Because of the order the quadtree is traversed (Session 2.1), the error is only evaluated for three edges of each quadrant: Center, Right and Bottom, as shown in Figure 9. These are combined and the local error for the level $l$ is $E_l = E_C + E_B + E_R$. Since each quadrant has four children, the immediate lower level error is given by $E_{l-1} = \max(E_{Q1}, E_{Q2}, E_{Q3}, E_{Q4})$, where $E_{Qi}$ is the level of each child quadrant. The quadrant error is then given by $E_Q = \max(E_l, E_{l-1}) + \varepsilon$, where and $\varepsilon$ is a small value sufficient to make the current quadrant error $E_Q$ higher than its children error.

This simple strategy guarantees that higher levels accumulate more error than lower levels, so that as the quadtree is traversed in a depth-first search, error tends to reduce. When such error, combined with viewer distance, becomes smaller than a user-defined threshold, the quadrant is enabled to be rendered. One must remember that neighboring resolution constraints, as presented in the previous session, must be taken into account. Lidstrom [6, 9] and Duchaineau [7] present in details more accurate error metrics, which can improve the visual accuracy of our approach.

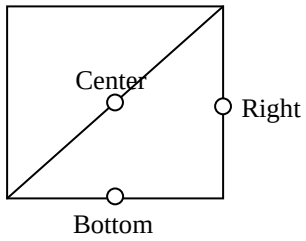



Figure 9: Edges that affluence mesh refinement

## 3 Results and Comparisons

**Performance Analysis:** In order to evaluate the proposed algorithm, a PC with a 2.6 GHz Pentium4 CPU, 512 MB system memory, and an NVidia GeForceFX 5800 GPU with 128MB video memory has been used. The proposed algorithm is compared with Ulrich [2], which source code is available in the internet. Many implementation aspects, including C and OpenGL programming tricks, as well a 512×512 height map data used on the tests, were borrowed from [2]. Table 1 shows the performance in frames per second (FPS) of both algorithms with different number of rendered triangles and culling configurations. In all tests, texture mapping was enabled and the rendering pipeline bottleneck [8] is on the CPU.

In general, the proposed algorithm presents an overall performance of at least 32% superior to [2] when culling is disabled. Both algorithms traverse the quadtree at least once, independent whether culling is enabled or not. However, when culling is enabled, some quadrants may be discarded. In such cases, [2] presents an increasing in performance proportional to the culled area, that corresponds to branches not visited on the second rendering traversal. It has been performed three different tests with different percentages of culled triangles.

Table 1: Performance comparisons among the amount of triangles rendered and frames per second (FPS) achieved. Culling is evaluated at the percentage of triangles removed from the graphical pipeline.

| | Ulrich [2] | | | | Pozzer et al. | | | |
|---|---|---|---|---|---|---|---|---|
| Triangles Rendered | Culling disabled | Culling enabled | | | Culling disabled | Culling enabled | | |
| | | 0% | 50% | 90% | | 0% | 50% | 90% |
| 20,000 | 207,8 | 206 | 148 | 54,2 | 281,4 | 267,7 | 180 | 58,0 |
| 50,000 | 96 | 95,5 | 70,2 | - | 137,4 | 131 | 87,6 | - |
| 100,000 | 53 | 53 | 37,7 | - | 70,1 | 64,7 | 45,4 | - |
| 150,000 | 37,2 | 37,1 | 25,4 | - | 51,8 | 48,4 | 36,6 | - |

When culling removes about 50% of the processed triangles, the performance difference drops to 20% in the worst case. The most extreme test processes an amount of 200,000 triangles, but with just 20,000 triangles (10%) not culled off. In such case, the performance of both algorithms becomes very similar. This culling setup leaves blanks in the table because the used height map did not support the required amount of triangles for higher configurations.

**Memory usage:** The runtime memory space requirements for the previous tests show that our approach uses 29.4 Mb. Ulrich [2] uses only 25.6 Mb, once his algorithm removes unnecessary nodes in a pre-processing step, according to some given error.

**Quadtree initialization:** In this benchmark it is compared the time each algorithm takes to start rendering the first frame. In this process texture loading is disabled. The proposed approach features a 5/16 times faster initialization. It takes 0.055 seconds, while [2] takes 0,934/0,275 seconds depending on whether removal of unnecessary nodes is considered or not. Such performance is based on an optimized non-recursive quadtree's construction that allocates a single vector buffer, for storing all nodes, instead of allocating individual nodes. This feature may be used as an efficient approach for constructing large terrains, organized as sets of tiled, smaller terrains. As the viewer moves in some direction, corresponding tiles can be dynamically and quickly loaded, while the ones that are not visible anymore can be discarded. This approach can be implemented without the use of sophisticated cache coherence, as presented by Lindstrom [9].

**Limitations:** One of the most noticeable limitations of the algorithm is the inability of changing abruptly the mesh resolution in order to best fit possible very high-contrasts terrains. As presented in session 3.2, it has been implemented three different patterns for mesh refinement. One can notice that such patterns only apply for fast resolution reduction. The same does not apply for resolution increasing. Figure 10 compares our approach with most common meshes from other related works. It can be noticed that the proposed approach takes

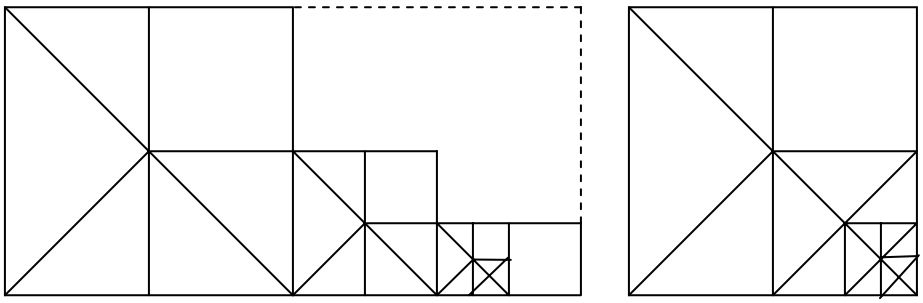


Figure 10: Mesh refinement speed

one more quadrant to achieve the best resolution, given the same neighboring details. It is important to observe that such limitation results from the fact that the mesh cannot present T-junctions. Apart from that limitation, it can benefit from local adaptivity, in contrast to [5] that makes use of this restriction in order to create data structures that avoid random-access traversals and hence consume less memory and are more efficient.

Figures 11 shows screen shots of the height field used on benchmarks, rendered at different resolutions. Figure 12 shows another example rendered with a detailed mesh, from a different viewpoint.

## 4 Conclusions and Future Work

In this work it has been presented a new algorithm for rendering terrain data. Its performance is based mainly on the ability to construct the irregular mesh with a single pass through the quadtree, taking into account both terrain geometry and viewer position. It also presents a fast data structure initialization, which is not covered in this paper, but can be observed in the source code. An interactive OpenGL/Glut demo application, with all source code and data files, can be downloaded at the site: http://www.icad.puc-rio.br/~pozzer/terrain.

The results show that the mesh is smoothly adapted according to terrain geometry. However, when interactively flying over the terrain, some small popping artifacts can be noticed. This may be resulted from the simplified error metric adopted for mesh simplification, or from the limited refinement speed the quadrants may have.

Main future issues include the definition of new refinement patterns in order to obtain a mesh that best fits local terrain geometry, especially when this presents high-contrasts. It is also important to implement a more accurate error metric for mesh simplification, to avoid popping effects, especially when rendering terrains at low resolutions.

## 5 Acknowledgments

This work is sponsored by CNPq and FINEP, through the Project VisionLab and individual research grants.

## 6 References

[1] Röttger, S., Heidrich, W., Slusallek, P., Seidel, H. *Real-Time Generation of Continuous Levels of Detail for Height Fields*, Universität Erlangen-Nürnberg, Technical Report 13/1997.

[2] Ulrich, T. *Continuous LOD Terrain Meshing Using Adaptive Quadtrees*, Gamasutra article, 2000, http://www.gamasutra.com/features/20000228/ulrich_01.htm (10-09-2003).

[3] Ulrich, T. *Chunked LOD: Rendering Massive Terrains using Chunked Level of Detail Control*,

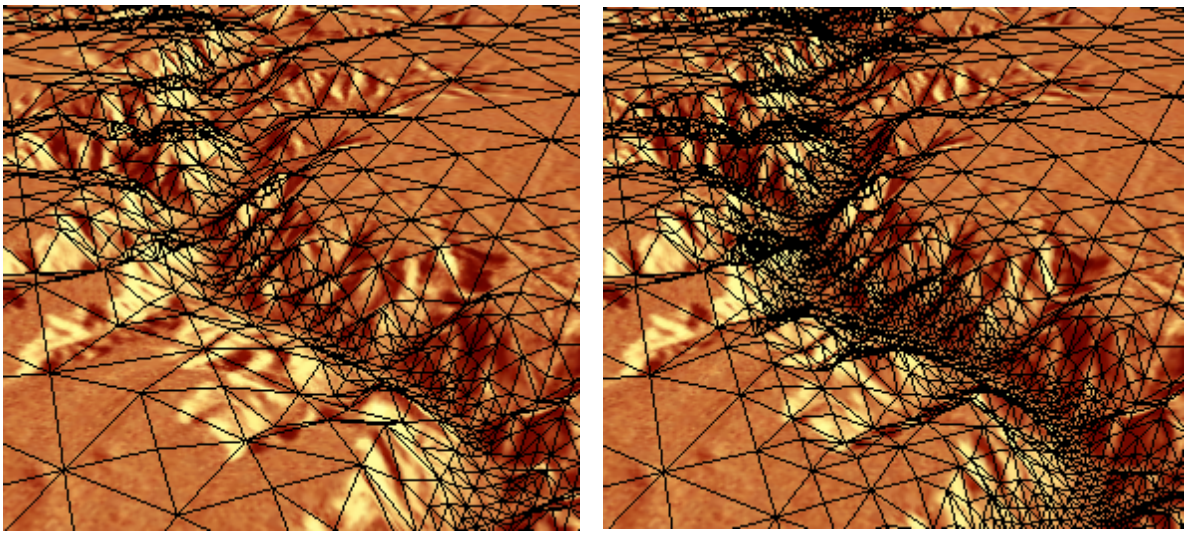


Figure 11: Height map rendered in low resolution (a) and in high resolution (b)

Course at SIGGRAPH 02, http://www.tulrich.com/geekstuff/chunklod.html (01-07-2004).

[4] Corpes, G. *Procedural Landscapes*, presentation at Game Developer Conference 2001.

[5] Losasso, F., Hoppe, H. *Geometry Clipmaps: Terrain rendering using nested regular grids*, ACM SIGGRAPH, pp. 769-776, 2004.

[6] Lindstrom, P., Koller, D., Ribarsky, W., Hodges, L. F., Faust, N., Turner, G. A. *Real-Time, Continuous Level of Detail Rendering of Height Fields*, In SIGGRAPH 96 Conference Proceedings, pp. 109-118, August 1996.

[7] Duchaineau, M., Wolinski, M., Sigeti, D. E., Miller, M. C., Aldrich, C., Mineev-Weinstein, M. B. *ROAMing Terrain: Real-time, Optimally Adapting Meshes*, Proceedings of the Conference on Visualization '97, pp. 81-88, October 1997.

[8] Moller, T., Haines, E. *Real-time rendering*, A. K. Peters Ltd, Natick, MA, 900 p., 2nd edition, July 2002.

[9] Lindstrom, P., Pascucci, V. *Terrain simplification simplified: A general framework for view-dependent out-of-core visualization*. IEEE Transaction on Visualization and Computer Graphics, 8(3), pp. 239-254, 2002.

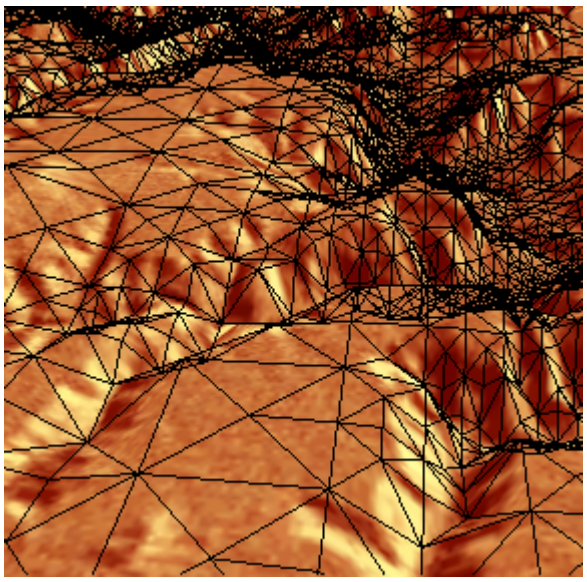


Figure 12: Height map rendered in high resolution