HyperBlock-QuadTIN:
Hyper-Block Quadtree based Triangulated Irregular Networks

Roberto Lario
Dpto. Arquitectura de
Computadores y Automática
Universidad Complutense Madrid
Spain

Renato Pajarola
Information & Computer
Science Department
University of California Irvine
USA

Francisco Tirado
Dpto. Arquitectura de
Computadores y Automática
Universidad Complutense Madrid
Spain

Abstract

Terrain rendering has always been an expensive task due to large input data models. Hierarchical mul-tiresolution triangulation and level-of-detail rendering algorithms over regular structures of grid digital elevation models have been widely used for interactive terrain visualization. The main drawbacks of these are the large cost of memory storage required and the possible over-sampling of high-resolution terrain models. Triangulated irregular networks (TIN) can reduce the amount of vertices at the expense of more complex and slower memory data access. We present a hyper-block quadtree based triangulated irregular networks approach, where the notion of vertex-selection is extended to block-selection. The hyper-block structure allows to store different pre-calculated triangulations. This reduces the vertex selection time per frame and removes the calculations needed to build the geometric rendering primitives (triangle strips) of the scene at the expense of a larger number of selected vertices. The presented approach shows a speed increment of 20% for high-quality terrain rendering with small screen-projection error thresholds.

Keywords: terrain rendering, height fields, quadtree.

1. Introduction

Real-time terrain visualization usually employs as input large data models with very high numbers of vertices. The very large data sizes generally exceed the capabilities of graphics hardware despite the advances experienced in this field over the last few years. Several solutions can be applied to simplify the geometric scene complexity and improve rendering performance.

View-dependent level-of-detail (LOD) algorithms present an efficient solution. These take into account the viewpoint location of the camera to satisfy a given screen-space error tolerance. In this way, points closer to the viewpoint have more weight in the LOD-selection and triangulation process. This vertex selection process is generally performed on a per-vertex basis. In contrast, in this paper we describe a view-dependent LOD algorithm that uses a block-selection approach. Our goal is to take advantage of a fast block-selection process and exploit pre-calculated triangle strips supported by our block-based data structures. These advantages reduce the LOD-selection and triangulation time, but increase the number of rendered triangles.

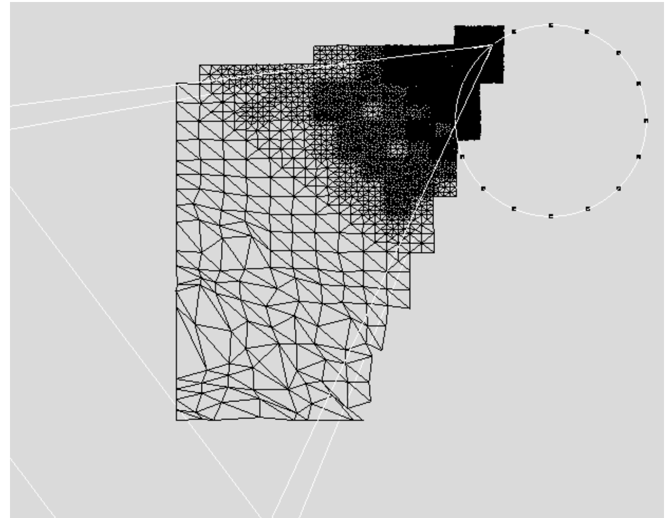


Figure 1. Top view of the view-frustum and the wire-frame HyperBlock-QuadTIN triangulation of a terrain.

Using the rendering capabilities of the latest graphics cards, we show that a terrain scene can be rendered faster using a block-based selection method at small screen-space error tolerances (equal or less than 1.5 pixels).

The input data models are QuadTIN-based [14] terrains, conforming to the restricted quadtree triangulation (RQT) properties [18]. Thus any irregular network previously pre-computed with QuadTIN can be used as input. Figure 1 shows an example of a HyperBlock-QuadTIN triangulation of a terrain.

Several terrain rendering methods have been proposed in the literature. Among the hierarchical methods, quadtree [15,19] and bintree [1,5,6,11,12] based approaches have been used extensively. Hoppe [9] and Pajarola et al. [14] present methods to render irregular terrain data sets. Other methods, like variable resolution 4-k meshes [21] exploit the advantages of subdivision connectivity to tesselate surfaces. Additional information about terrain rendering can be found in [2,4,5,8,10, 13,16,17]. In particular, the cached geometry rendering approach presented in [11] is very similar to a block-based LOD selection and rendering approach. Other block-based implementations could be found in [3,21] where precomputed-blocks of geometry information are used.

The remainder of the paper is organized as follows. Section 2 introduces the restricted quadtree triangulation. Section 3 briefly discusses the QuadTIN method. The HyperBlock-QuadTIN approach is presented in detail in section 4. Finally, section 5 and 6 end the paper with experimental results and conclusions about this work.

2. Restricted Quadtree Triangulation

The restricted quadtree triangulation [18] (RQT) is an adaptive, hierarchical LOD-triangulation algorithm for grid digital terrain elevation models. Every elevation point is assigned to a level in the quadtree hierarchy. The basic recursive quadtree subdivision and triangulation is performed in two steps as shown in Figure 2.

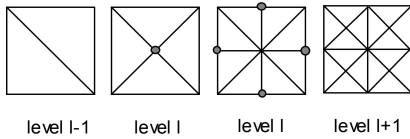


Figure 2. Recursive quadtree subdivision and triangulation.

Cracks can occur in the triangulated surface from unrestricted adaptive subdivision and triangulation as shown in Figure 3. To avoid this situation, the RQT subdivision is constrained by the restriction that adjacent quadtree blocks differ by at most one level in the hierarchy. Then the triangulation is adjusted to resolve cracks as outlined below.

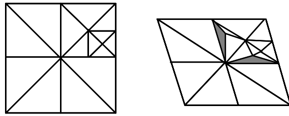


Figure 3. Nonrestricted quadtree triangulation. Cracks are shown with grey colour in the right.

An efficient method to avoid cracks was introduced in [12], this applies a dependency relation as illustrated in Figure 4. Each vertex on level l specifies two others on the same level as in Figures 4 b) and d), or on level l-1 as in Figures 4 a) and c).

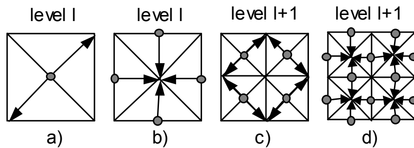


Figure 4. Dependency relation of the RQT. The center vertices in a) and c) depend on the inclusion of two corners of their quad region. The boundary edge midpoints in b) and d) depend on the center vertices of adjacent quad regions.

For every selected point, its two dependency-points must also be selected. This method not only avoids cracks but also ensures a triangulation that can be represented by one single triangle strip.

3. QuadTIN

QuadTIN [14] is an efficient quadtree-based triangulation approach for irregular triangulated networks. It provides fast quadtree-based adaptive triangulation, view-dependent LOD-selection and real-time rendering.

Basic quadtree-based triangulation methods are applicable only to regular grid input datasets. In contrast, QuadTIN presents an efficient quadtree-based triangula-tion approach to irregular input point sets with improved storage cost and feature adaptive sampling resolution while preserving a regular quadtree multiresolution hierarchy over the irregular input data set. It achieves this by inserting a small number of Steiner points to the input data set (generally less than 25% of the initial data points). This technique allows a single triangle strip representation of the terrain data. Although the quadtree hierarchy is not balanced, it conforms to the restricted quadtree constraints. Additional information such as geometric approximation error, bounding spheres and normal cones are calculated and stored in each quadtree node to be used for view-dependent LOD-triangulation and rendering.

4. HyperBlock-QuadTIN

The HyperBlock-QuadTIN approach generates a tree structure of blocks that store different triangulation levels, thus the name hyper-block. The construction process takes a QuadTIN file as input (see Figure 5) and traverses the hierarchy up to a certain level that we call *hyperLevel* at which hyper-blocks are built that encompass the remaining levels in that sub-tree.

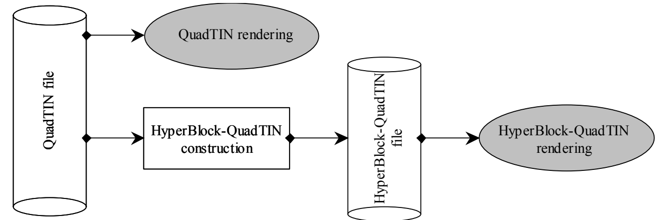


Figure 5. HyperBlock-QuadTIN diagram.

Figure 6 illustrates hyper-blocks generated at different hyperLevels (1 and 2) for a given quadtree of height 3. If hyperLevel is equal to 0 then the entire quadtree is represented by one single hyper-block.

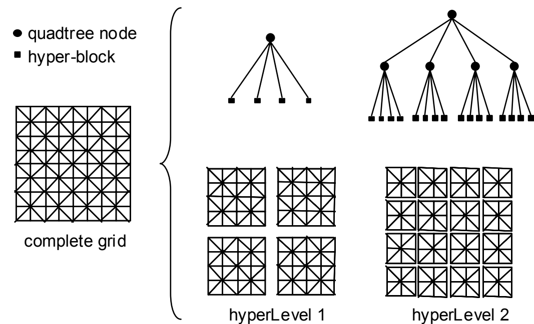


Figure 6. Complete grid and the corresponding hyper-blocks for hyperLevels 1 and 2. HyperLevel 0 coincides with complete grid.

The main advantange of Hyper-QuadTIN is that it uses of vertex-blocks instead of individual vertices to determine the LOD-triangulation. This permits to precalculate static triangle strips for hyper-blocks. Also the LOD-selection time can be improved by the block-wise vertex selection.

4.1. Construction

The construction can be divided into four steps. The first works on the input tree, the remaining three work with hyper-blocks.

- Generation of quadtree hyper-blocks.
- Basic block-level construction.
- Extended block-level construction.
- Assignment of block-errors.

Hyper-block generation: Taking the QuadTIN hierar-chy as input, this stage builds a new quadtree with depth equal to hyperLevel. At this level it encompasses the remaining levels from the QuadTIN hierarchy into a hyper-block data structure (see also the following section on the hyper-block data structure). It is possible that some branches in the QuadTIN hierarchy have a depth which is smaller than hyperLevel, see also Figure 7. This special situation is dealt with by introducing a special hyper-block that has just one level (and one triangulation). Such special hyper-blocks can hang from any level smaller than hyperLevel in the quadtree. Normal hyper-blocks hang from the last level in the quadtree, which is exactly the hyperLevel.

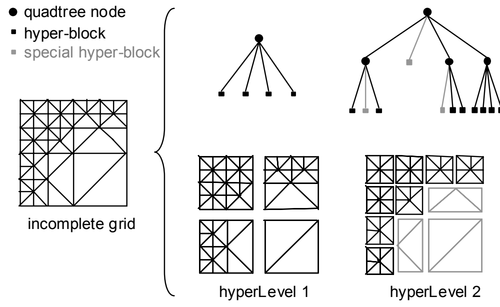


Figure 7. Incomplete grid and the corresponding blocks for hyperLevels 1 and 2. Notice the special hyper-blocks due to the non-balanced QuadTIN hierarchy which hang from arbitrary levels.

Basic block-level construction: This stage calculates the number of internal levels of each hyper-block. Due to the non-balanced QuadTIN input hierarchy, every hyper-block has a different number of levels. In every block-level of a hyper-block a triangle strip is built and stored as an index-buffer following the scheme shown in Figure 8. We call these basic configurations. At rendering time, according to the view-dependent LOD criteria, a particular basic triangulation level of a hyper-block is selected.

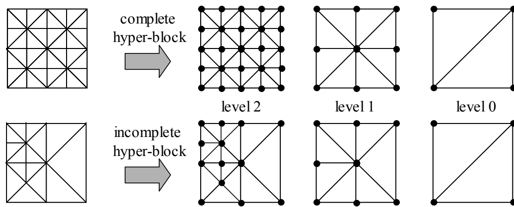
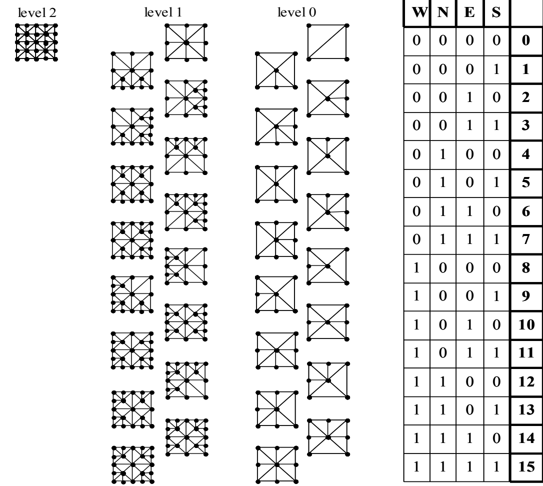


Figure 8. Basic block-level configurations of two differ-ent hyper-blocks.

Extended block-level construction: In order to join adjacent hyper-blocks without cracks in the triangulation we restrict the selected triangulation level of adjacent hyper-blocks to differ by at most one level. Thus we have to consider all combinations of adjoining basic configurations of one level difference. In fact, we always extend the appropriate border of the level l block to conform to the triangulation of the adjacent block with level l+1.

A quad has four borders: south, east, north and west (S,E,N,W). Consider a level l in a hyper-block, vertices of level l+1 can be inserted on any border from an adjacent block. The combination of the four border triangulations (taking into account all possibilities) gives fifteen additional possible triangulations, which we call extended configurations. Figure 9 shows in detail all triangulations of a hyper-block with three levels. This approach allows to always join hyper-blocks that differ by at most one level without cracks in the triangulation. For example, a hyper-block with basic configuration level l+1 (Figure 9, case 0 in the border table) can be joined in the south by a hyper-block at level l with extended-north triangulation (Figure 9, case 4 in the border table).



W	N	E	S	
0	0	0	0	**0**
0	0	0	1	**1**
0	0	1	0	**2**
0	0	1	1	**3**
0	1	0	0	**4**
0	1	0	1	**5**
0	1	1	0	**6**
0	1	1	1	**7**
1	0	0	0	**8**
1	0	0	1	**9**
1	0	1	0	**10**
1	0	1	1	**11**
1	1	0	0	**12**
1	1	0	1	**13**
1	1	1	0	**14**
1	1	1	1	**15**

Figure 9. Border table and the corresponding triangula-tions for levels 0, 1 and 2. W, N, E and S correspond to the west, north, east and south border configuration. The sixteen cases can be expressed by a 4-bit integer. Configuration 0 is the basic triangulation.

Assignment of block-error: In each hyper-block, different LOD approximation errors are assigned to the different basic triangulation configurations (levels). These level-errors are computed as the maximum geometric approximation error of the basic triangulation at this level (the infinite-norm of the vertical distance of unused vertices to the triangulated surface).

4.2. Data structures

Three simple data structures are used in the HyperBlock-QuadTIN construction process and for rendering:

- Tree
- HyperBlock
- InfoLevel

Tree: It is a quadtree data structure with a depth equal to hyperLevel. The field *hyper* stores a HyperBlock pointer that points to a regular hyper-block, a special hyper-block,

or is zero if no block is referenced. Bounding sphere data is also stored to perform view-frustum culling.

```
struct Tree
{
   Tree* parent;
   Tree* child_sw; //south-west son
   Tree* child_se; //south-east son
   Tree* child_ne; //north-east son
   Tree* child_nw; //north-west son
   HyperBlock *hyper;
   float x,y,z,r; //center and radius of bounding sphere
}
```

HyperBlock: This structure contains the information of every hyper-block. The fields *selectedLevel* and *borderType* will be used in the rendering process to select the level and decide the border configuration of Figure 9. The field *isSeleted* is used as to know if a hyper-block is selected in a certain frame, i.e. it passes the frustum culling test. The field *isSpecial* specifies normal or special hyper-block types (see section 4.1). Information about adjacent hyper-blocks is contained in the *s, e, n, w* fields which are used to adjust the border configuration according to Figure 9 in the rendering process as described in Section 4.3.

```
struct HyperBlock
{
   bool isSpecial; // true if hyper-block is special case
   bool isSeleted; // boolean selection variable
   int numLevel;     // number of levels of hyper-block
   InfoLevel* level;// pointer to InfoLevel data
   int selectedLevel;// selected level in current frame
   int borderType; // =0..15 config. type (see Figure 9)
   HyperBlock *s;    //adjacent south hyper-block
   HyperBlock *e;    //adjacent east hyper-block
   HyperBlock *n;    //adjacent north hyper-block
   HyperBlock *w;    //adjacent west hyper-block
}
```

InfoLevel: The basic and extended triangulations of every hyper-block level are store in this data type. The sixteen triangle strip configurations (basic plus fifteen extended triangulations) shown in Figure 9 are generated during the construction process and stored in *strip* (with the lengths stored in *numStrip*). The field *error* contains the maximum geometric approximation error of the vertices that belong to the respective basic configuration level.

```
struct InfoLevel
{
   int numStrip[16];
   int* strip[16];
   float error;
}
```

4.3. HyperBlockQuadTIN rendering

The rendering process can be divided into three stages:

- Hyper-block and basic block-level selection.
- Basic block-level adjustment.
- Extended block-level selection.

Hyper-block and basic block-level selection: To select the hyper-blocks within the view-frustum, the tree of hyper-blocks is recursively traversed top-down. The bounding sphere of each node is used to perform the view-frustum test (see Figure 10). The distance from each view-frustum pyramid plane to the bounding sphere center is calculated and compared with the bounding sphere radius. If greater (supposing plane-normals of view-frustum pyramid are oriented inside-out) then any descendant nodes and hyper-blocks are out of the view-frustum and this node is dismissed.

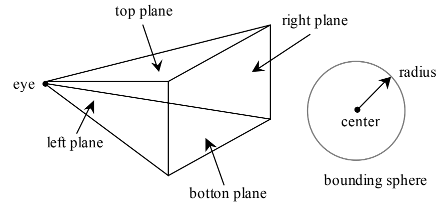


Figure 10. View-frustum and bounding sphere of a tree node.

When a hyper-block is selected, the level-errors are projected onto the screen as shown in Figure 11 and compared with a given image-space error tolerance. The smallest level with projected error smaller than the given error tolerance specifies the basic block-level for this hyper-block.

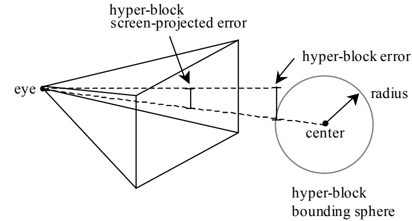


Figure 11. View-frustum, hyper-block error and the screen-projected error.

Basic block-level adjustment: Once the hyper-block levels are chosen, this stage removes block-level differences greater than one between adjacent hyper-blocks by increasing the level of the lower-level neighbors (see Figure 12 b).

Extended block-level selection: At this point, hyper-block levels are set and the basic configuration is assumed. The last adjustment uses the extended border configuration table (Figure 9) to guarantee a crack-free triangulation across hyper-blocks as shown in Figure 12 c).

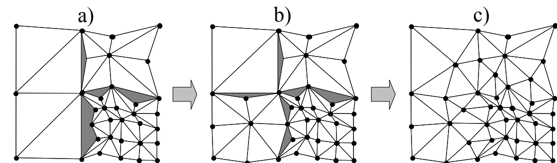


Figure 12. Rendering stages: a) Basic block-level selection. b) Basic block-level adjustment. c) Extended block-level selection.

Figure 12 shows a simple example of a terrain consisting of four hyper-blocks. In Figure 12 a) one can see the four basic configurations corresponding to the selected block-levels. In Figure 12 b) basic block-level adjustment is carried out. Note that the lower-left hyper-block increases its level during this step. The cracks finally disappear in the extended block-level selection in Figure 12 c). The level transitions can be observed in more details in Figure 13.

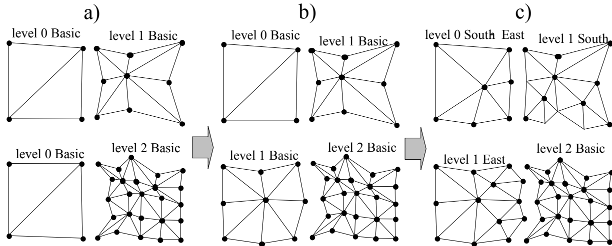


Figure 13. Rendering stages and level transitions of hyper-blocks.

5. Results

The performance analysis was tested on a 2.4GHz Pentium 4 with 1GB RDRAM and NVIDIA GeForce4 Ti 4400 graphics card. In all the scenes a 45º vertical field-of-view camera followed a circular path around the center of the terrain data models. Experimental results were averaged over 2500 frames in a window of 1024 x 768 pixels. OpenGL extensions are used to cache the height-field vertices on the video card. The terrain we used to perform the analysis is the Puget Sound data set. The complete grid (2049 x 2049) has 4198401 points. The preprocess with Terra software [7] (error = 5.0 meters) and QuadTIN [14] gives a terrain with 822099 vertices. The simplification process dramatically reduces the number of points with small loss of quality. This data model is not a very large field and can be stored completely in video memory, allowing the use of fully cached geometry. No cached-geometry-memory manager was considered to deal with larger terrain data models. Experimental rendering results were obtained for different hyperLevels. The following Figures show the comparison between the basic QuadTIN rendering (individual per-vertex LOD selection) and the HyperBlock-QuadTIN rendering (block-based LOD selection) with the three best hyperLevels (in this example hyperLevels 7, 8 and 9). The HyperBlock construction process consumed less than 15 seconds for any hyperLevel setting used with the given terrain data.

The most illustrative data is the frame-rate comparison (see Figure 14). The best frame rate in the 0 to 1 pixel error tolerance range is achieved by the Hyper-Block approach with a hyperLevel of 8. Only at higher error tolerances (error=1.4 pixels) QuadTIN is faster. Note that HyperBlock outperforms QuadTIN for high level-of-detail, i.e. very low pixel tolerances, for all three considered hyperLevels. The frame-rate improvement with one pixel error tolerance is about 21% for hyperLevel 8. Other studied terrains gave similar results.

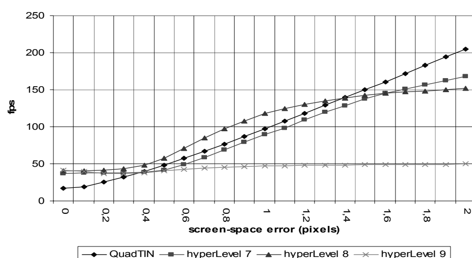


Figure 14. Puget Sound (822099 vertices) frame-rate.

Larger hyperLevels mean smaller block sizes and more numerous blocks. More blocks involve a more expensive selection process because more blocks have to be selected and more borders have to be adjusted in the final rendering stage. On the other hand, smaller blocks in general also imply fewer rendered triangles (Figure 15).

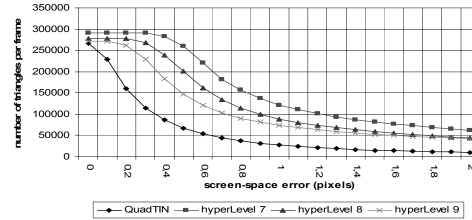


Figure 15. Number of triangles per frame (Puget Sound).

If we see the selection time (Figure 16) we can conclude that the LOD selection time remains almost constant because the number of hyper-blocks selected in the view-frustum for every frame mainly depends on the hyperLevel. Larger hyperLevels generate a greater number of hyper-blocks. In fact this is exactly what a block-based LOD selection approach is designed for. Instead of an expensive per-vertex LOD selection process the block-based LOD selection dramatically reduces the time required to generate the LOD triangulation. Clearly, the block-based LOD selection and triangulation process is largely independent of the error tolerance. In comparison we can see that the per-vertex QuadTIN selection time decreases exponentially with the error tolerance, or is proportional to the number of selected triangles (Figure 15).

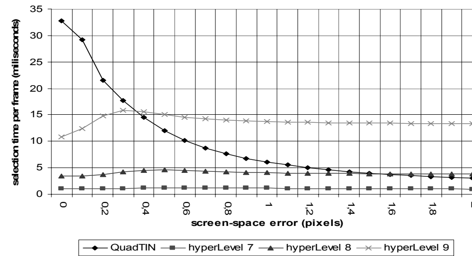


Figure 16. Selection time per frame for the Puget Sound data set (822099 vertices). For QuadTIN, the selection time also includes the time needed to build the triangle strip.

The display time itself (rendering triangles) is largely proportional to the number of triangles (Figure 15). In fact, both approaches compared here use triangle strips and cached geometry. Only LOD selection and triangulation cost depend on the used data structures and algorithms. Therefore, a block-based approach can only save time by a simplified LOD selection and triangulation approach. On the other hand, the higher-number of rendered triangles generally work against any block-based approaches. This is exactly what we can observe in the given experiments and we further discuss our findings in Section 6.

6. Discussion

HyperBlock-QuadTIN provides very good results for rendering large terrains at very high visual quality and

shows up to 20% performance improvement in the presented experiments. As shown in our tests, the block-based LOD vertex selection and triangulation approach is superior compared to a per-vertex LOD selection process. It provides a quasi-constant view-dependent LOD triangle mesh generation cost. Despite the disadvantage in higher triangle counts, given a fixed triangles-per-second rendering rate, the block-based approach is faster overall up to a certain break-even image-space error tolerance after which the vertex-based approach continues to be slightly better. This break-even point depends on the ratio between CPU and GPU (graphics processor) speed and usage.

Due to its efficient triangulation process (constant time LOD selection and pre-constructed triangle strips), the block-based approach makes much less use of the CPU and only slightly more use of the GPU than the per-vertex approach. The speed of GPUs currently increases at a faster rate than the performance of main CPUs. Due to this observation we can conclude that the break-even error tolerance below which the block-based approach is most advantageous will continue to increase. Therefore, block-based triangulation and rendering approaches will be more and more effective as GPU power increases faster than CPU performance. Furthermore, with increasing GPU speeds smaller and smaller error tolerances and higher triangle counts can be handled at interactive frame rates which also tend to favor block-based approaches.

We presented our block-based triangulation approach as an extension to QuadTIN which is a hierarchical multiresolution triangulation approach for irregular terrain point sets. This is no restriction of the approach since the hyper-block generation and rendering outlined in Section 4 works without modification also with regular quadtree hierarchies over grid-digital terrain elevation models. The choice of block-sizes remains to be a difficult problem. Deciding which is the optimal hyperLevel for a given terrain data set is difficult. We can observe a tradeoff between rendering and triangulation cost that at the current stage has to be exploited for different data sets. We want to explore how effective block-sizes can automatically be determined in future work.

7. Acknowledgement

Special thanks to the US Geological Survey for the digital elevation models shown in this paper and GVU Center at Georgia Tech for the preparation of the textured Puget Sound data set. Also, we would like to thank the University Complutense of Madrid and the Spanish research grant TIC 2002-750 for their support of the principal investigator of this research project.

References

[1] Laurent Balmelli, Serge Ayer, and Martin Vetterli. Efficient algorithms for embedded rendering of terrain models. In *Proceedings IEEE ICIP 98*, pages 914–918, 1998.

[2] Mark de Berg and Katrin Dobrindt. On levels of detail in terrains. In *11th Symposium on Computational Geometry*, pages C26–C27. ACM, 1995.

[3] Glenn Corpes. Procedural Landscapes. *Game Developer Conference 2001*.

[4] Leila De Floriani and Enrico Puppo. Hierarchical triangulation for multiresolution surface description. *ACM Transactions on Graphics*, 14(4):363–411, 1995.

[5] Mark Duchaineau, Murray Wolinsky, David E. Sigeti, Marc C. Miller, Charles Aldrich, and Mark B. Mineev-Weinstein. Roaming terrain: Real-time optimally adapting meshes. In *Proceedings IEEE Visualization 97*, pages 81–88, 1997.

[6] Williams Evans, David Kirkpatrick, and Greg Townsend. Right-triangulated irregular networks. *Algorithmica*, 30(2):264–286, March 2001.

[7] Michael Garland and Paul S. Heckbert. Fast polygonal approximation of terrains and height fields. Technical Report cmu-cs-95-181, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1995.

[8] Thomas Gerstner. Multiresolution compression and visualization of global topographic data. Technical Report 29, Institut fur Angewandte Mathematik, Universitat Bonn, 1999.

[9] Hugues Hoppe. Smooth view-dependent level-of-detail control and its application to terrain rendering. In *Proceedings IEEE Visualization 98*, pages 35–42. Computer Society Press, 1998.

[10] Reinhard Klein, Daniel Cohen-Or, and Tobias Huttner. Incremental viewdependent multiresolution triangulation of terrain. In *Proceedings PacificGraphics 97*, pages 127–136. IEEE, Computer Society Press, 1997.

[11] Joshua Levenberg. Fast view-dependent level-of-detail rendering using cached memory. In *Proceedings IEEE Visualization 2002*, pages 259–265. Computer Society Press, 2002.

[12] Peter Lindstrom, David Koller, William Ribarsky, Larry F. Hodges, Nick Faust, and Gregory A. Turner. Real-time, continuous level of detail rendering of height fields. In *Proceedings SIGGRAPH 96*, pages 109–118. ACM SIGGRAPH, 1996.

[13] Peter Lindstrom and Valerio Pascucci. Visualization of large terrains made easy. In *Proceedings IEEE Visualization 2001*, pages 363–370. Com-puter Society Press, 2001.

[14] Renato Pajarola, Marc Antonijuan, and Roberto Lario. QuadTIN: quadtree based triangulated irregular networks. In *Proceedings IEEE Visualization 2002*, pages 395–402, 2002.

[15] Renato Pajarola. Large scale terrain visualization using the restricted quadtree triangulation. In Proceedings *IEEE Visualization 98*, pages 19–26,515, 1998.

[16] Renato Pajarola. Overview of quadtree-based terrain triangulation and visualization. Technical Report UCI-ICS-02-01, I&C Science, University of California Irvine, 2002.

[17] Enrico Puppo. Variable resolution terrain surfaces. In Proceedings of the *8th Canadian Conference on Computational Geometry*, pages 202–210, 1996.

[18] Hanan Samet. The quadtree and related hierarchical data structures. *ComputingSurveys*, 16(2):187–260, June 1984.

[19] Ron Sivan and Hanan Samet. Algorithms for constructing quadtree surface maps. In *Proc. 5th Int. Symposium on Spatial Data Handling*, pages 361–370, August 1992.

[20] Thatcher Ulrich. Rendering Massive Terrains using Chunked Level of Detail Control. Course at *SIGGRAPH 2002*.

[21] Luiz Velho and Jonas Gomes. Variable resolution 4k meshes: Concepts and applications. *Computer Graphics Forum*, 19(4):195–214, 2000.