Modern Algorithms for Real-Time Terrain Visualization on Commodity Hardware

Radek Bartoň
Department of Computer Graphics and Multimedia
Faculty of Information Technology
Brno University of Technology
ibarton fit.vutbr.cz

Keywords: computer graphics, rendering, real-time, level of detail, out-of-core, DEM, heightmap

Abstract

The amount of input data acquired from a remote sensing equipment is rapidly growing. Interactive visualization of those datasets is a necessity for their correct interpretation. With the ability of modern hardware to display hundreds of millions of triangles per second, it is possible to visualize the massive terrains at one pixel display error on HD displays with interactive frame rates when batched rendering is applied. Algorithms able to do this are an area of intensive research and a topic of this article. The paper first explains some of the theory around the terrain visualization, categorizes its algorithms according to several criteria and describes six of the most significant methods in more details.

Introduction

Surface of our planet is an omnipresent entity in our lives. Scientists and engineers always studied its shape as it takes an important role in construction planning, hydrology, cartography and other fields. Also a lot of people take pleasure in bread views on landscape in their leisure time. These are the reasons for precise and correct terrain modeling and high-quality visualization using modern technologies in GIS, virtual realities, simulations or computer games.

Recent elevation datasets of the Earth [1][2] reaches nearly 0.5 TB sizes at 1 arc second resolution. Ortophoto images are usually more precise and more accurate datasets can be expected in the near future. Handling this vast quantity of data is a tremendous task when interactivity is required despite the advances in performance of state of the art general purpose hardware.

Fortunately some techniques for dealing with this problem were developed and they are constantly improved. This article intends to explain some basic concepts and to present few

representatives of the most modern algorithms for large and huge terrain visualization in more detail.

Important Theory

A few accustomed terms are used in general the *level of detail* or terrain visualization specialized literature. Let us explain some of them that are used in or closely related to the following text and that may not be familiar to the reader:

DEM

Digital elevation model or *DEM* is a discreet representation of terrain surface in a general form. In transferred meaning, available terrain datasets can be referred to DEMs.

Input Data Scale

From a terrain visualization point of view, we can distinguish among the following scales of the input domain:

- **Small terrains** – Commonly occupy tens of MB of memory space and cover for example hundreds of km^2 with 5 meters grid size.
- **Large terrains** – Tens of GB and for example millions of km^2 with 10 meters grid size.
- **Huge (massive) terrains** – Usually spread out on planetary-scale area (for example hundreds millions of km^2 at 10 meters grid size) and require from hundreds to thousands of GB.

Heightmap

It is the most simple but also the most used format of terrain representation. X and Y coordinates can be stored implicitly, saving two thirds of memory, but it usually occupies the most space since the memory requirements are not adaptable to the terrain shape. See figure 1 for an example.

TIN

Triangulated irregular network (*TIN*) is different designation for a mesh with an arbitrary topology (see figure 2). Its advantages is in its precise adaptation to a terrain shape, and thus it usually takes the lowest amount of memory, and can model caves and overhangs. On the other hand, it demands significant processing power when working with it.

RTIN

Right-triangulated irregular network (*RTIN*) is a compromise between efficiency of the *heightmap* and scalability of the *RTIN* (figure 3). It can be represented implicitly using a binary

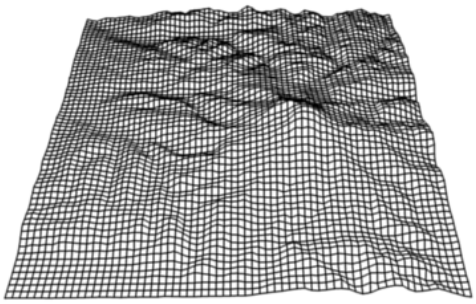

Figure 1: Different types of input data representation – Heightmap.

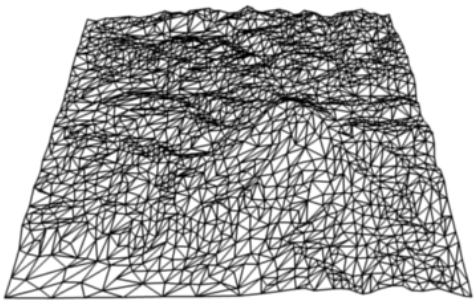

Figure 2: Different types of input data representation – TIN.

tree or a quad-tree.

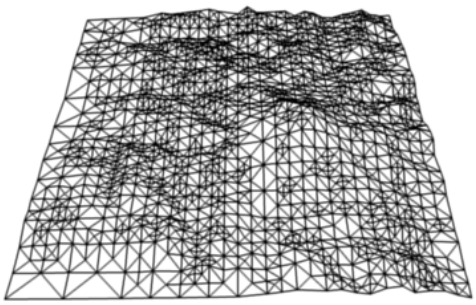

Figure 3: Different types of input data representation – RTIN.

Level of Detail

To achieve interactive frame rates, a number of rendered triangles has to be reduces. A discrete amount of this reduction is called the *level of detail*. Because of the properties of the

perspective projection, the *level of detail* has to be different for different parts of displayed objects. When the change in *level of detail* simplification is gradual over the mesh surface, we speak about a *continuous level of detail*.

Pixel Error

The most important task for each terrain *level of detail* algorithm is decision if a given vertex or a group of vertices is visually so significant that it should be visible in the final triangulation. To solve this, many different error metrics has been developed. The problem is illustrated in figure 4 from which the notion of the *pixel error* is apparent – a geometric error projected onto a screen with a certain pixel resolution using a perspective projection.

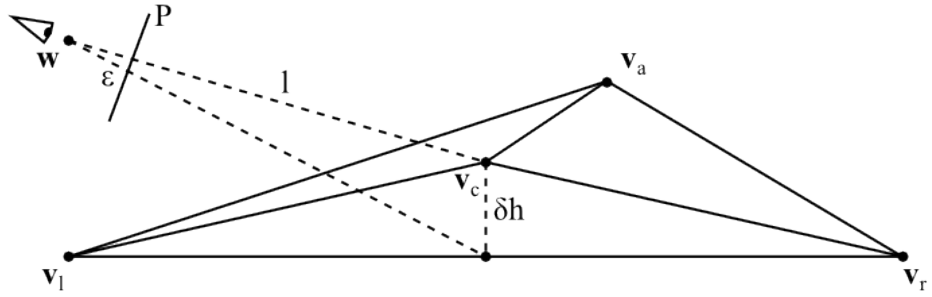

Figure 4: Demonstration of a *pixel error*. Merging of two triangles ($\mathbf{v_c}$, $\mathbf{v}_l$, $\mathbf{v}_a$) and ($\mathbf{v}_c$, $\mathbf{v}_a$, $\mathbf{v}_r$) into a single triangle ($\mathbf{v}_a$, $\mathbf{v}_r$, $\mathbf{v}_l$) by removing a vertex $\mathbf{v}_c$ which distance to a viewpoint $\mathbf{w}$ is l produces the *pixel error* projected onto a plane P. δh is the difference between $\mathbf{v}_c$ and the middle of a hypotenuse of the new triangle.

View-Frustum Culling

When rendering with a perspective projection, the visible space has a shape of truncated pyramid – the *view frustum*. Any object outside this volume may be discarded from the rendering pipeline. It is usually done by a point-wise test against six half-spaces combined with some heuristic.

Geomorphing

If the allowed display error is for the performance reasons higher than one pixel, sudden changes in the terrain shape can be seen when a tile is switched from one resolution to another. In such cases, the *geometrical morphing* or just *geomorphing* of the shape is employed. It is usually done by a linear interpolation of higher *level of detail* vertex positions to a height of an appropriate edge midpoint in the lower *level of detail* geometry. This interpolation is bidirectional and when the limit position is reached, particular *level of detail* can be exchanged for a lower or a higher one, respectively. The sources of the interpolation factor value can be different. Examples are: time, frame number or viewer's distance to the center of the tile.

Restricted Quad-Tree

Restricted quad-tree is a data structure and an associated algorithm [3][4] for terrain heightmap simplification that generates a *RTIN* triangulation which has no gaps or T-vertices. It defines relationships between heightmap vertices that have to be included in the triangulation if the vertex they are dependent on is included too.

Shader

It is a computer graphics program executed on a GPU written in a specialized programming language. There is a common trend to move all tasks of the terrain visualization algorithms to the GPU. Currently, only the *vertex shader*, processing vertices of rendered triangles, and the *pixel shader*, manipulating fragments of the resulting image, are utilized in the algorithms.

Modern Terrain Visualization Algorithms

After some basic terminology has been consolidated, this section will give an overview on the six most significant terrain visualization algorithms and their variants. The descriptions are as thorough as necessary to understand the basic concepts of the algorithms, nevertheless interested reader might seek for the referenced papers for further theoretical and implementation details.

ROAM 2.0

Hwa et al. modernized [5] a well-known but already obsolete *Real-time Optimally-Adapting Meshes* (*ROAM*) algorithm [6], therefore their contribution is referred to as *ROAM 2.0*. A similar approach is also the *RUSTiC* algorithm [7] but *ROAM 2.0* is more advanced. It facilitates batched rendering, which is modern graphics hardware optimized to, by exchanging ROAM's individual triangles with triangular patches with a regular 4-8 mesh topology [8][9]. The original triangle binary tree is replaced with a more optimized data structure based on a quad-tree of triangle *diamonds*.

Each *diamond* in this tree represents a vertex that is shared as an apex of four neighboring right-angled triangles. In contrast to the regular quad-tree, all non-root diamonds in the diamond quad-tree have two parents, so this data structure should rather be called DAG. Figure 5 shows this data structure more explanatory.

A single dual queue refinement of the geometry tiles is extended with additional merge and split queues for the texture tiles. This means that geometry and texture *levels of details* are selected independently. There is another priority queue for scheduling of the texture objects creation since this is a costly operation performed in each frame. An out-of-core data streaming indexing scheme is supported using a Sierpinski space filling curve (figure 6) both for the geometry and the texture patches organization on an external storage. The triangle strip indices within the tile are also ordered using this space-filling curve to utilize the GPU vertex cache efficiently.

Run-time refinement exploits temporal coherence provided by the dual priority queues. Each frame, the priories are recomputed using *view-frustum* culling for the current viewing parameters and then they are balanced with split and merge operations to achieve some error value

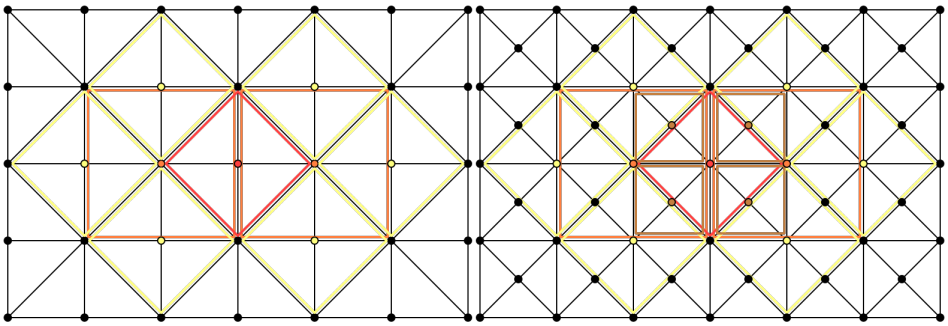

Figure 5: Demonstration of two *levels of detail* of a 4-8 triangle mesh in a ROAM 2.0 algorithm. A *diamond* – four triangles sharing an apex vertex – (depicted in red color) has four children diamonds in the higher resolution level (the light brown ones), two parents (in orange) in a lower resolution level and six siblings (neighbors) (the yellow diamonds). The colored vertices are those the appropriate diamonds (the colored squares on the figure) are assigned to.

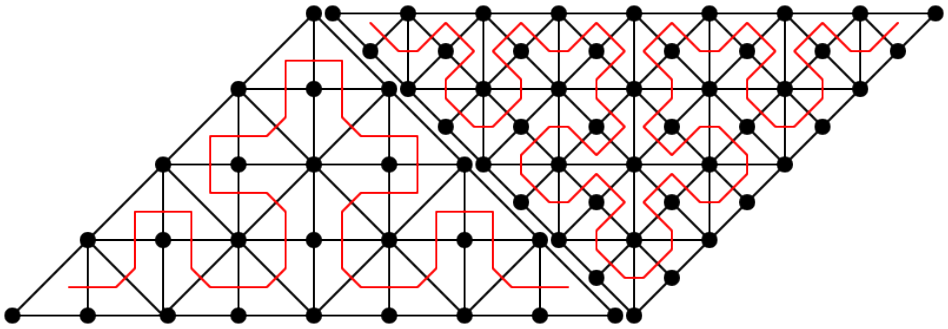

Figure 6: Example of a Sierpinski space filling curve on two consecutive levels of detail of a 4-8 triangle mesh.

at the fronts of the queues. Procedure can be stopped when specific error measure or defined patch count is reached. In this way, the constant frame rate can be guaranteed.

Geometrical Mip-Mapping

A very simple method (in its original form) is the *Geometrical Mip-Mapping* algorithm [10]. The terrain domain is divided into regular tiles with $2^n + 1; n = 1, 2, \ldots$ vertices on each side. A quad-tree of axis aligned bounding boxes is then constructed, as is shown in figure 7, with only the leaf nodes containing actual geometry and topology in several precomputed resolutions. The vertices can be stored in vertex buffers and topology is precomputed to index buffers.

During run-time, the hierarchy is traversed from the top to the bottom and tiles' bounding boxes are tested against the *view frustum*. If the bounding box is completely outside the frustum or a leaf node is reached, descending for the current subtree stops. The visible leaf nodes are recorded in a list of tiles to render and an error metric, which selects the appropriate *level of detail* for the tile, is computed.

Because of the different resolution of neighboring tiles in the resulting triangulation produced

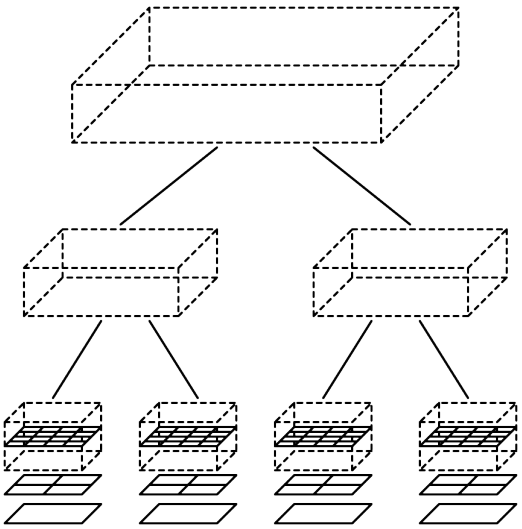

Figure 7: One dimensional scheme of a tile quad-tree in the *Geometrical Mip-Mapping* algorithm. While non-leaf nodes contain only bounding boxes, leaf nodes contain bounding boxes, full resolution geometries as well as other subsampled geometry levels.

by the prior procedure, some gap filling and T-vertex removal operations have to be done. There are many ways how to handle the problem but the method proposed in the original paper is that only an interior part of the tile's geometry is rendered using precomputed index buffers and border strips, connected to the tiles with a higher resolution, are rendered using triangle fans as depicted in figure 8. This figure also illustrates an example of the mip-mapped tiling.

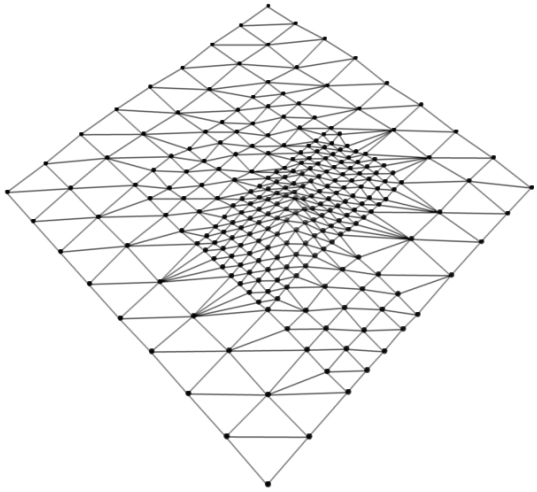

Figure 8: Filling of gaps between neighboring tiles of a different *level of detail* using triangle fans. Topology of lower resolution tiles is adapted to higher resolution tile borders.

Used view-dependent error metric is reformulated to a form, where for each *level of detail* of a tile, there is a distance from which this particular level should be used. A residual fraction of this distance is used as the *geomorphing* interpolation factor.

In addition to this elemental concepts, many further improvements and extensions to the

basic framework were elaborated. Among the first, Poyart et al. [11] discussed how to deal with non-square terrains of an arbitrary size and how to minimize preprocessing cost using a *heightmap* sample skipping method for multiresolution tile rendering.

Another work [12] explains how to optimize *geomorphing* using vertex *shader* and also mentions utilization of a *potentially visible set* (PVS) for simple occlusion culling.

The paper [13] only presents some implementation notes of the former algorithm but uses different topology for the tiles – a longest-edge bisection triangulation.

Pouderoux and Marvie [14] use tiled toroidal buffer of tiles for efficient data streaming over the network. The buffer is in every frame asynchronously updated on a ring by ring basis from the center to the border as much as the viewer's speed allows. The *level of detail* selection is performed depending on a given polygon budget and data availability which allows for a constant frame rate control. On the other hand, authors omitted the quad-tree and the *view-frustum* test is computed for each tile in the buffer. Also the *geomorphing* is not being considered anymore and a proposed gap-filling method relies on a view-projected textured plane under each tile, similar to their shadows (see figure 9).

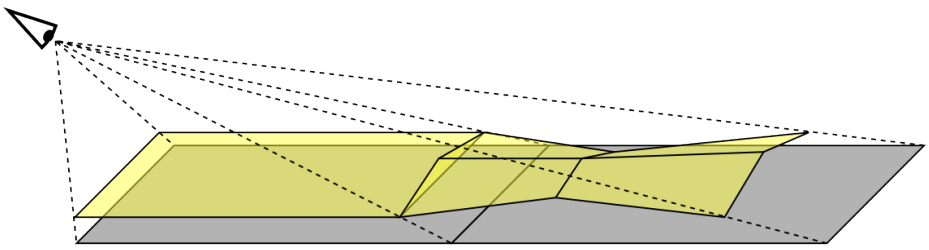

Figure 9: Visual masking of gaps between neighboring tiles of a different *level of detail* using textured and view-projected planes.

Lastly, Schneider and Westextitann [15] proposed another significant improvements. They precompute the geometry of the tiles using a *restricted quad-tree* simplification algorithm [3] with an exponentially increasing object-space geometry error and then build triangle strips using any high-quality triangle strip generation method. This allows progressive transmission of vertex coordinates and minimizes the memory transfers using a vertex coordinate and index quantization. This is controlled by a GPU memory manager with the last recently used and tightest fit caching strategy.

Chunked LoD

Another algorithm called *Chunked LoD* [16] differs from the previously mentioned mainly in the way how the quad-tree stores the terrain geometry and consequently in slightly different traversal of the tree. The root contains a square tile of the whole terrain with the geometry of an arbitrary topology simplified to an error value (a *chunk*) and its bounding box. Its children are four square tiles each covering a quarter of its parent's extent simplified with a half of the error value of the parent and so on as figure 10 describes.

The tree is traversed from the top to the bottom and each time a projected error of a node satisfies a pixel error threshold or a bounding box is outside the *view frustum*, the descending

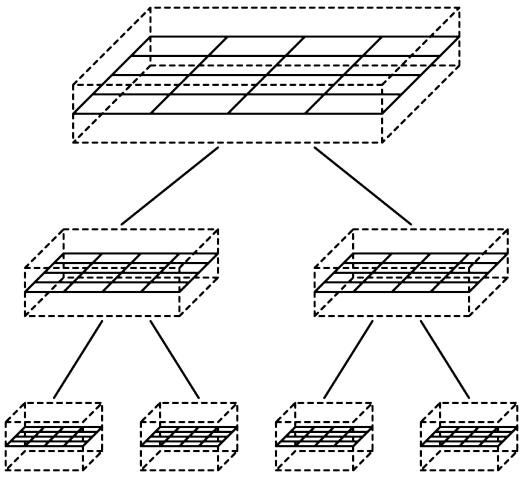

Figure 10: One dimensional schema of a tile quad-tree in a *Chunked LoD* algorithm. Each node contains a bounding box of the tile and its geometry. Object-space error of the geometry is decreasing and size of the tiles is shrinking exponentially from the root to the leafs.

stops and the tile is rendered or discarded, respectively. Otherwise, non-leaf node's children are visited. The resulting triangulation is depicted in a figure 11. The author proposes four different methods to stitch neighboring chunks together like to restrict the simplification at tile borders, use flanges or skirts around the tile or to fill gaps with additional triangles but prefers to use the skirts which is also apparent in the figure 11.

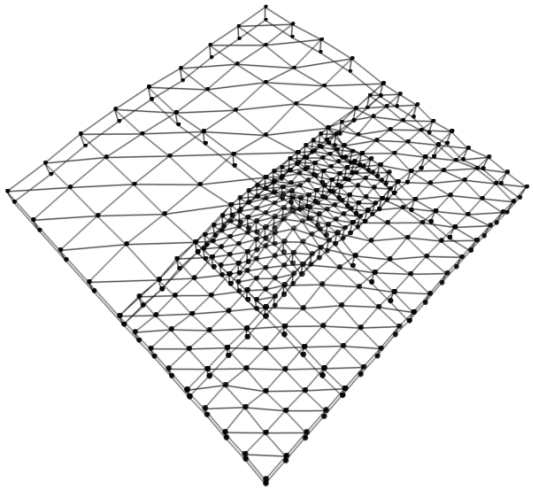

Figure 11: Example of tiling of a *Chunked LoD* algorithm. Covering of gaps between neighboring tiles of a different *level of detail* is done by rendering vertical skirts around each tile's border.

The *view-frustum* culling and the *geomorphing* is also supported. The geometry within the *chunks* can be irregular and can be heavily compressed for efficient streaming, on the other hand, dynamic terrain modifications are impracticable.

Geometrical Clipmaps

This approach, introduced in 2004 [17] and improved in 2005 [18], uses another analogy from texturing – *clipmaps* [19]. They are an extension to the *mipmaps*, offering support to incrementally page texture data into toroidal buffers and so allowing a feasible work even with huge textures.

Consider few *mipmap* levels with exponentially growing size in both dimensions as on figure 12. The *clipmap* is then set of *mipmap* levels cropped using certain but fixed value (yellow parts) – a *clipmap size*. Trimmed parts (orange) are not present in the main memory and stay stored on an external drive ready for paging in when a point of interest called a *clipmap center* moves.

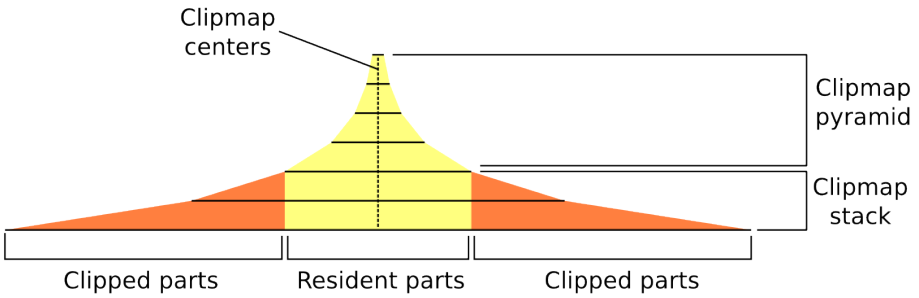

Figure 12: Clipmaps construction from mipmaps by cropping to *clipmap size*.

Clipmap levels can be separated into two groups: The ones that has *clipmap size*^{2} dimensions, since they were made form bigger *mipmaps*, forming a *clipmap stack* and *clipmap pyramid* made from original *mipmaps* that were smaller. Both groups requires special treatment but *clipmap pyramid* is a rather static and equivalent to the top of the *mipmap* pyramid.

Geometrical clipmaps use this concept for managing of huge *heightmaps* and textures, however *clipmap* levels are stored on the GPU in two dimensional vertex textures and the complete heightmap pyramid is kept in the main memory in a highly compressed form. Square tiles of the geometry generated from the *clipmap* levels with the same grid resolution (the *clipmap size*) are nested within each other, occupying exponentially growing geometrical space as shown in figure 13. It can be derived [17] what is the recommended *clipmap size* for a given screen resolution and an used field of view to achieve sub-optimal triangle count and to prevent aliasing artifacts.

During run-time, four different sets of clipmap areas are maintained. First are the so called *clip regions* (black *cr X* regions in figure 14 a)) that represent extent of each *clipmap* region data that are present in the GPU memory. Second are square *active regions* (red *ar X* regions on the 14 a)) of *clipmap size* side size centered at the actual viewer's position. Those are desired new boundaries for the *clipmap* levels for the current frame. Missing data (yellow and orange areas on the figure 14 a)) are copied, if possible, to the toroidal buffers before the rendering. *Clipmap* buffers are updated in an order from the coarse to the fine until a given time quota is reached. Unfinished memory transfers (orange areas on the figure) are left and *active regions* are cropped accordingly, to form *crop regions* (red dashed areas on the figure 14 a) and black solid regions on the figure 14 b)). Lastly, *render regions* are created as differences between *crop region* for a current *clipmap* level and a *crop region* for the next finer

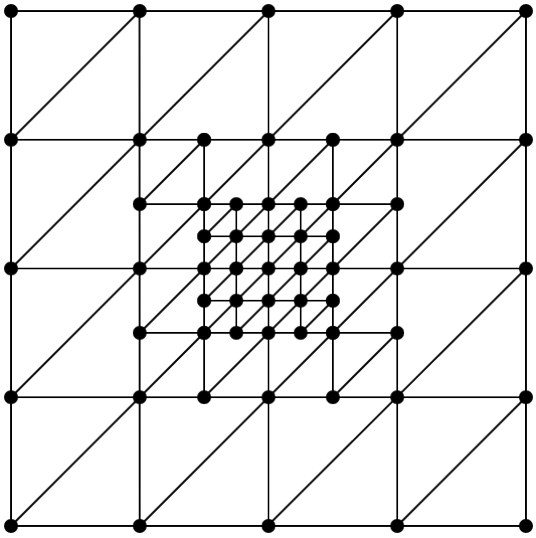

Figure 13: Nested square *clipmap* tiles of the *Geometrical Clipmaps* algorithm. Practical tile resolution is between 255 and 2047 vertices and not 5 as depicted.

level (colored areas on the figure 14 b)). At the end of the frame the *crop regions* become *clip regions* for the next frame.

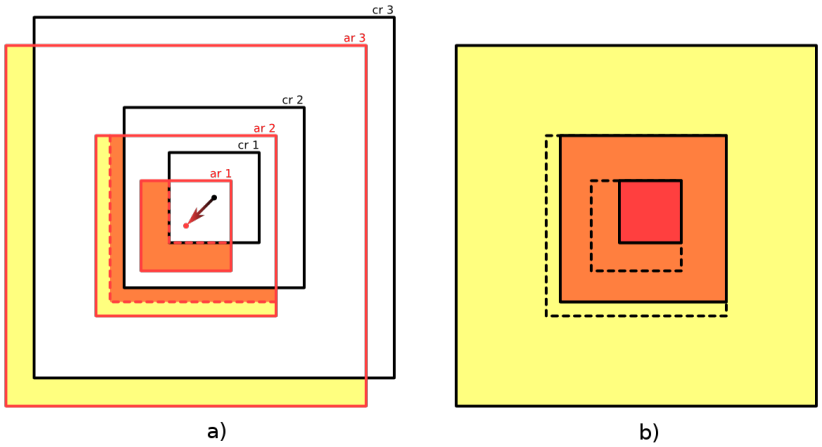

Figure 14: Regions of geometrical *clipmaps*. a) *Clip regions* (*cr X*, black squares) are borders of data present in the graphics memory. *Active regions* (*ar X*, red squares) are borders of data that need to be loaded for the current frame. *Crop regions* (dashed red rectangles) are the *active regions* trimmed to data extent that was actually loaded in time. b) *Render regions* (colored areas) are disjoint differences between *crop regions* of consecutive *clipmap* levels. Dashed squares depicts extent of the *render regions* if the data would be fully updated.

As mentioned before, updates of square buffers are performed toroidally which avoids unnecessary memory transfers. Figure 15 describes this process. If the *clipmap center* moves, new data have to be loaded at the buffer's border. In a simple case of the figures 15 b) and 15 c), only three block copies are necessary to perform this. On the other hand, the figure 15 d)

presents a worst case scenario when five block transfers are needed. The *clipmap* data are then read (or written) using modulo addressing. Amount of data for each *clipmap* update depends on speed of the user over the terrain and on the geometrical size of the *clipmap*. Greater user's distance to the surface can also disable updates of too fine *clipmap* levels.

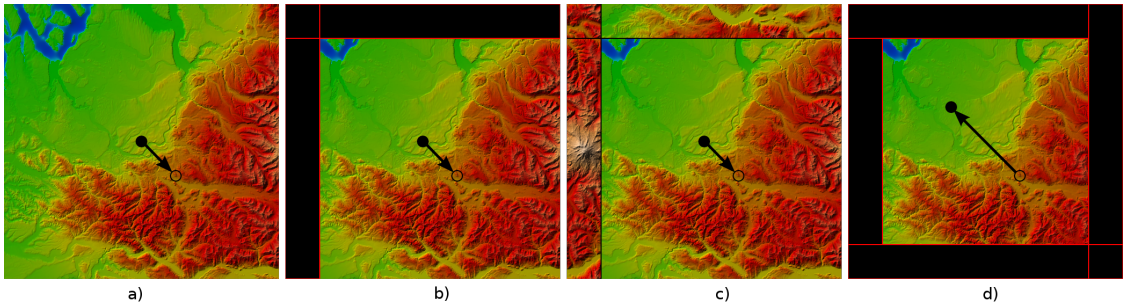

Figure 15: Toroidal updates of *clipmap* buffers. a) Initial state of the buffer before an user moves from the black point to the empty point. b) After the user moves, the black parts of the buffer has to be reloaded. c) They are filled with three blocks toroidaly. Addressing is then done using modulo function. d) Worst case scenario. When the user moves backwards over a buffer's edge, five blocks has to be updated. Source of data: USGS. [20].

To off-load geometry processing from the CPU to the GPU, the *render regions* could be constructed using two-dimensional tiles precomputed as several index and vertex buffers and hight values are then added in a geometry shader from the *heightmap* texture. Figure 16 shows an example of such tiling. We need five types of the tiles for ring *render regions* and one another when a central *render region* is handled apart. Nevertheless, this implies restriction on *clipmap* update procedure: *Active regions* have to be fully updated or empty to deal with their static geometry. On the other hand such blocks of geometry can be used for easy-to-implement *view-frustum* culling that brings significant rendering speedup. Because the shape of the tiles is identical up to the translation and scale, geometry instancing could be used to further improve the overall rendering performance.

The last thing that needs to be explained is how to fix gaps incurred at *clipmap* borders. First rule to avoid this is that the number of vertices in each geometrical *clipmap* has to be odd, as in figures 13 and 16, to match odd vertices of a finer *clipmap* to all vertices of a coarser *clipmap*. Second part of the solution is that some transition areas at the borders are established where height values are interpolated between their original position and the position in a neighboring coarser *clipmap* similar to *geomorphing* in the *Geometrical Mip-Mapping* algorithm. And lastly, zero area triangles around *clipmap*'s perimeter hast to be rendered to fill possible artifacts caused by floating point computation inaccuracy.

Spherical extension to the basic *geometrical clipmaps* concept also exits [21]. Spherical *clipmaps* practically form an integral half of a globe mesh always oriented towards the viewer's eye. They are made of concentric rings with an exponentially growing density of parallels from the equator to the pole. Figure 17 presents four of such *clipmaps*. Nice consequence is that no special care has to be taken to stitch neighboring *clipmaps* together. A disadvantage compared to the planar version is that an input rectangular *heightmap* has to be sampled with the coordinates transformation in a vertex shader since the geometry is parametrized using spherical coordinates.

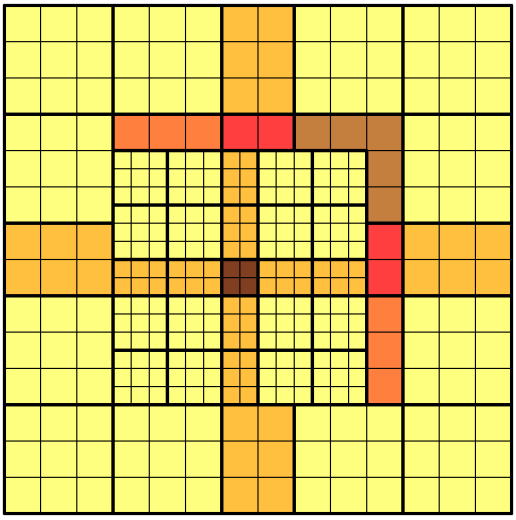

Figure 16: Tiling of a geometrical *clipmap*. Tiles depicted with same color share their index and vertex buffers. Size (4x4 vertices) of the most frequently used type (yellow tiles) is chosen to optimally utilize vertex caching.

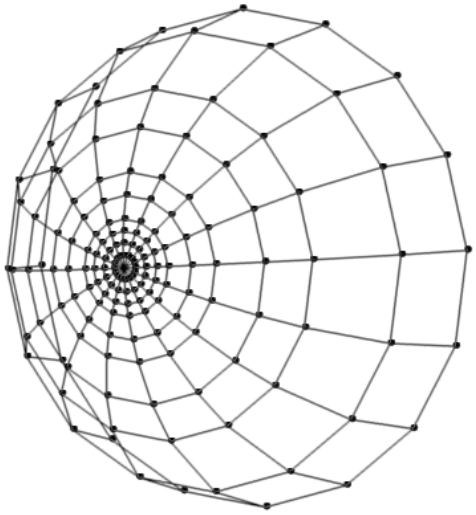

Figure 17: A half-globe of four spherical *clipmaps* with an exponentially growing density of parallels.

BDAM

Batched Dynamic Adaptive Meshes (*BDAM*) [22][23][24][25] employs a triangular tile topology similarly as *ROAM 2.0*. The main idea exploits the property of RTINs that a triangle can be connected to another triangle from the same level, one level higher triangle through its hypotenuse or one level lower triangle through one of its catheti. A terrain triangulation can be then represented as a set of triangular tiles from a binary tree hierarchy simplified to some error measure which is shared between possibly neighboring triangles from adjacent tree levels along tile borders and which is smoothly grading inside the tile. This is depicted on a figure 18 where the first three images are three levels of the tile binary tree and the last

image is an example of such triangulation. Maximal geometrical error for each level of the three decreasing exponentially.

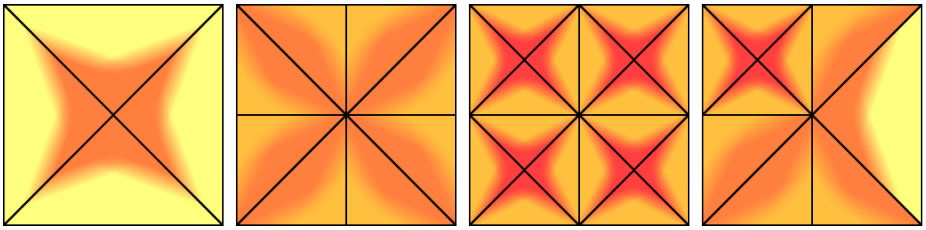

Figure 18: Three consecutive levels of a binary tree of tiles of a *BDAM* algorithm and a resulting triangulation made of them.

Using this organization, no further care has to be taken to fill the gaps in the triangulation because every tile can be connected just to the tiles simplified to same error value and thus with seamlessly matching borders. The geometry inside the tile can have an arbitrary topology and it can be in a preprocessing step adaptively refined and optimized with high-quality decimation and triangle strip generation algorithms and then compressed for its effective transfer during rendering.

A square terrain texture is represented with a quad-tree since this is a more suitable representation for it but in order to better assign appropriate texture quad-tree nodes to geometry binary tree nodes, the binary tree had to be split into two halves. A hierarchy of nested bounding error volumes is also built in which serves for view-dependent error refinement and *view-frustum* culling. A full description of these volumes is quite complex [3] but, in a nutshell, they denote the distance from which certain tile should be visible. If the viewer is outside this sphere, the tile and all its descendants do not have to be included in the triangulation.

The refinement is driven by texture quad-tree descending and when a suitable texture tile is chosen, it is bound to hardware as a texture object. Then the refinement continues in both geometry binary trees from the nodes which correspond to the texture node until a geometry error measure is met. The geometry tiles are then rendered for the whole texture tile so it minimizes the necessary context switches.

Also this algorithm has some extensions. A *planetary-scale* dataset support has been elaborated in [23] (see figure 19) and an advanced wavelet-based compressing schema has been discussed in [25].

Seamless Patches

An elegant solution to the tile connectivity problem is presented in the paper *Seamless Patches for GPU-Based Terrain Rendering* by Livny et al. [26]. While they maintain squared tile topology for efficient texturing, they separate each tile, referred to as *patches*, into four triangular sub-tiles, referred to just as *tiles*. The left part of figure 19 demonstrates that.

The *tile*-level *level of detail* selection is decided at the *patch* borders so that neighboring *patches* always share the same number of vertices and are connected seamlessly. *Tiles* within the *patch* are stitched with precomputed strips (right part of the figure 19) which number is dependent on the number of possible *levels of detail* and can be computed as:

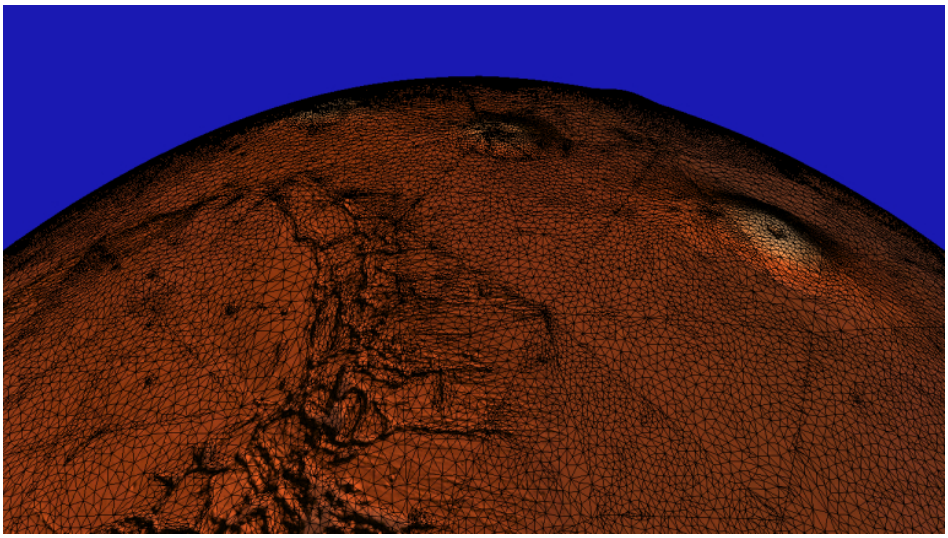

Figure 19: Example of *planetary-scale* dataset visualization using P-BDAM algorithm. Source [25].

$k = \frac{(l+1)!}{2*(l-1)!}$

where l is the actual number of the *tile*'s *levels of detail*.

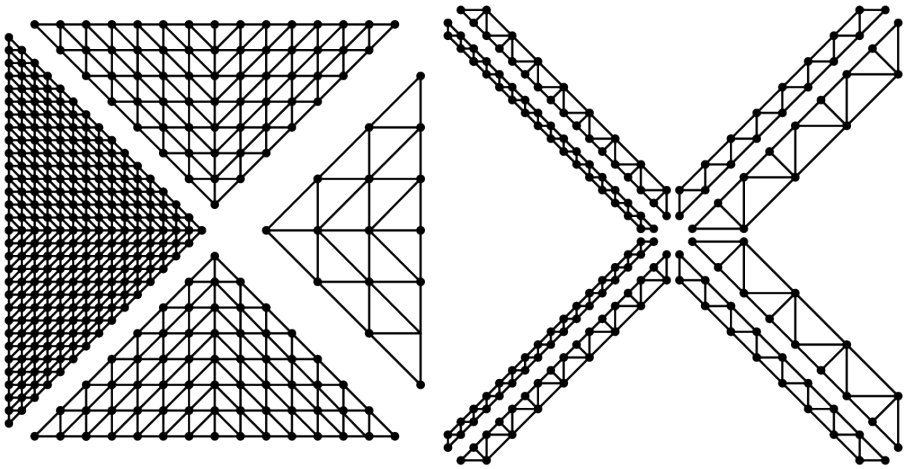

Figure 20: Triangular *tiles* of a seamless *patch* (left) and adequate stitching *strips* (right).

For *patch*-level *level of detail* selection and *view-frustum* culling an implicit hierarchy of *patch* bounding boxes is constructed and it is traversed top-down during run-time in the same manner as in the *Chunked LoD* algorithm. The exceptions is that all quad-tree nodes do not contain any geometry but only *patch* position and dimensions. These values are used to transform and instantiate precomputed uniform size two dimensional vertex buffer objects. Vertex shader displacement mapping supplies the third dimension.

Proposed error metric [26], based on projected geometric length of the *patch*'s edge, implicitly ensures that for neighboring *patches* matching *tiles* are selected. Number of *tile levels of details* determines maximal local adaptivity within the patch. Practically only few precomputed *tile*

resolutions are used and the local adaptivity of the surface is originated from the quad-tree subdivision. Furthermore, higher numbers of allowed *tile* resolutions imply a higher branching factor of the *patch* hierarchy to guarantee correct connectivity because only the maximal possible resolution of a larger *patch* and the minimal possible resolution of a smaller patch can lie side by side. See the figure 20 for examples.

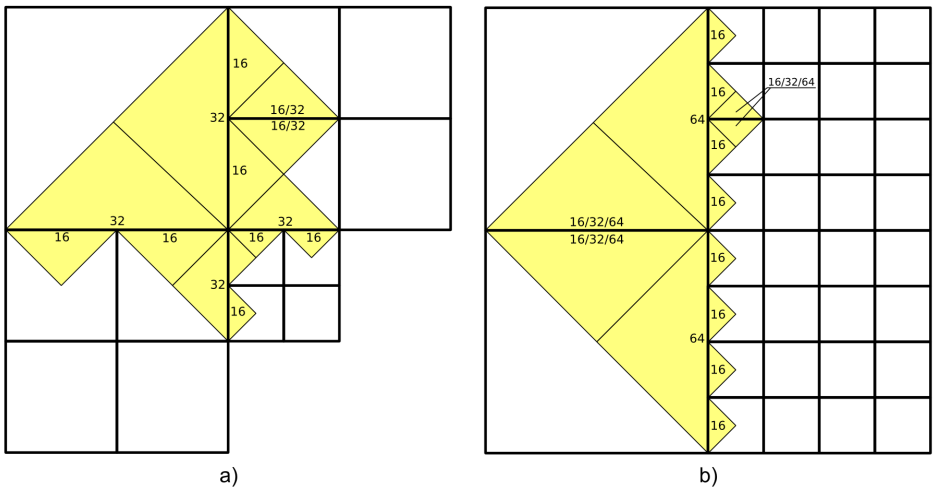

Figure 21: Tile resolution assignment in the *Seamless Patches* algorithm. a) In this case the number of *levels of detail* of the *tiles* is 2 (e.g. 16 and 32 vertices) and the branching factor is 2x2. Only 1:1 *patch* neighboring with 16:16 or 32:32 vertex resolution or 1:2 *patch* neighboring with 16:32 vertex resolution is allowed. b) In this case the number of *levels of detail* of the *tiles* is 3 (e.g. 16, 32 and 64 vertices) and the branching factor is 4x4. Only 1:1 *patch* neighboring with 16:16, 32:32 or 64:64 vertex resolution or 1:4 *patch* neighboring with 16:64 vertex resolution is allowed.

Conclusion

Many quite different approaches to the massive terrain visualization problem have been presented in the paper but a lot of them have the same origins in the previous, today obsolete, algorithms. Also other state of the art algorithms exist but they have been omitted due to extent limitations.

Overall algorithms performance is a quantity that is hard to express but it can be measured in millions of triangles per second rendered on the reference graphics equipment. We refer the reader to the original papers to see the values the authors claim about their implementations on their equipment. Objective evaluation of the algorithms' performance can be considered as out of scope for this paper or as a subject for the future work.

References

1. Jet Propulsion Laboratory, California Institute of Technology, "Shuttle radar topography mission", July 2010.
   Available at: http://www2.jpl.nasa.gov/srtm/

2. Earth Remote Sensing Data Analysis Center, "ASTER GDEM", July 2010. Available at http://www.gdem.aster.ersdac.or.jp

3. R. Pajarola, "Large scale terrain visualization using the restricted quadtree triangulation", *Proceedings of the Conference on Visualization '98*, pp. 19–26, IEEE Computer Society Press Los Alamitos, CA, USA, 1998.

4. R. Pajarola, M. Antonijuan and R. Lario, "QuadTIN: quadtree based triangulated irregular networks", *VIS '02: Proceedings of the Conference on Visualization '02*, pp. 395–402, 2002.

5. L. M. Hwa, M. A. Duchaineau and K. I. Joy, "Real-time optimal adaptation for planetary geometry and texture: 4-8 tile hierarchies," *IEEE Transactions on Visualization and Computer Graphics*, vol. 11, no. 4, pp. 355–368, 2005.

6. M. A. Duchaineau, M. Wolinsky, D. E. Sigeti, M. C. Miller, M. B. Mineev-Weinstein and C. Aldrich, "ROAMing terrain: Real-time optimally adapting meshes", *IEEE Visualization*, pp. 81–88, 1997.

7. A. A. Pomeranz, "Roam using surface triangle clusters (RUSTiC)". Diploma thesis, University of California at Davis, 2000.

8. W. Evans, D. Kirkpatrick and G. Townsend, "Right triangular irregular networks", University of Arizona, Tucson, AZ, USA, 1997.

9. W. Evans, D. Kirkpatrick and G. Townsend, "Right-triangulated irregular networks", *Algorithmica*, vol. 30, no. 2, pp. 264–286, 2001.

10. W. H. de Boer, "Fast terrain rendering using geometrical mipmapping", 2000. Available at: http://www.flipcode.com/articles/article_geomipmaps.pdf

11. E. Poyart, P. Frederick, R. D. B. Seixas and M. Gattass, "Simple real-time flight over arbitrary-sized terrains," *Workshop Brasileiro de GeoInformatica*, p. 5, Citeseer, 2002.

12. D. Wagner, "Terrain geomorphing in the vertex shader", *ShaderX2, Shader Programming Tips and Tricks*, pp. 1–12, 2003.

13. B. D. Larsen and N. J. Christensen, "Real-time terrain rendering using smooth hardware optimized level of detail", *Journal of WSCG*, vol. 11, no. 1, p. 8, 2003.

14. J. Pouderoux and J. Marvie, "Adaptive streaming and rendering of large terrains using strip masks", *Proceedings of the 3rd International Conference on Computer Graphics and Interactive Techniques in Australasia and South East Asia*, p. 306, 2005.

15. J. Schneider and R. Westermann, "GPU-friendly high-quality terrain rendering", *Journal of WSCG*, vol. 14, no. 1-3, pp. 49–56, 2006.

16. T. Ulrich, "Rendering massive terrains using chunked level of detail control", *SIGGRAPH Course Notes*, vol. 3, no. 5, 2002.

17. F. Losasso and H. Hoppe, "Geometry clipmaps: terrain rendering using nested regular grids", *International Conference on Computer Graphics and Interactive Techniques*, pp. 769–776, ACM New York, NY, USA, 2004.

18. A. Asirvatham and H. Hoppe, "Terrain rendering using GPU-based geometry clipmaps", *GPU Gems*, vol. 2, no. 2, pp. 27–46, 2005.

19. C. C. Tanner, C. J. Migdal and M. T. Jones, "The clipmap: a virtual mipmap", *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*, pp. 151–158, ACM New York, NY, USA, 1998.

20. USGS and The University of Washington, "Puget sound terrain", July 2010. Available at: http://www.cc.gatech.edu/projects/large_models/ps.html

21. M. Clasen and H.-C. Hege, "Terrain rendering using spherical clipmaps", *Eurographics/IEEE-VGTC Symposium on Visualization* (T. Ertl, K. Joy, and B. Santos, eds.), 2006.

22. P. Cignoni, F. Ganovelli, E. Gobbetti, F. Marton, F. Ponchio and R. Scopigno, "BDAM – batched dynamic adaptive meshes for high performance terrain visualization", *Computer Graphics Forum*, vol. 22, pp. 505–514, Sept. 2003.

23. P. Cignoni, F. Ganovelli, E. Gobbetti, F. Marton, F. Ponchio and R. Scopigno, "Planet-sized batched dynamic adaptive meshes (P-BDAM)", *IEEE Transactions on Ultrasonics, Ferroelectrics and Frequency Control*, pp. 147–154, 2003.

24. E. Gobbetti, P. Cignoni, F. Ganovelli, F. Marton, F. Ponchio and R. Scopigno, "Interactive out-of-core visualisation of very large landscapes on commodity graphics platform", *Lecture Notes in Computer Science*, pp. 21–29, 2003.

25. E. Gobbetti, F. Marton, P. Cignoni, M. Di Benedetto and F. Ganovelli, "C-BDAM – compressed batched dynamic adaptive meshes for terrain rendering", *Computer Graphics Forum*, vol. 25, pp. 333–342, Sept. 2006.

26. Y. Livny, Z. Kogan and J. El-Sana, "Seamless patches for GPU-based terrain rendering", *The Visual Computer*, vol. 25, pp. 197–208, Mar. 2008.