Eurographics Italian Chapter Conference (2008)
V. Scarano, R. De Chiara, and U. Erra (Editors)

Fractal Compression Approach for Efficient Interactive Terrain Rendering on the GPU

Ugo Erra[1], Vittorio Scarano[2] and Davide Guida[2]

[1]Università degli Studi della Basilicata, Italy
[2]Università degli Studi di Salerno, Italy

Abstract

This paper describes an efficient technique for the rendering of large terrain surfaces. The technique is based on a simple rings structure: a sequence of concentric rings at different resolutions and centered on the viewer's position. Each ring is represented by a set of patches at identical resolutions. Rings near the viewer have a finer resolution than the rings further from the viewer. At runtime, the patches within the rings change resolution based on the viewer's position. The GPU decodes in real time height maps encoded by a fractal compressor from which sample the height component of the terrain. Since adjacent patches of different rings can disagree on the resolution of common edge GPU stitches the meshes in order to avoid any cracks or degenerate triangles. The rendered meshes ensure the absence of cracks that may cause the appearance of visual artifacts. In addition, a tile manager support is evaluated in order to maintain terrain datasets on disk storage avoiding a costly load of the entire datasets into the memory.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Picture and Image Generation I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism I.3.7 [Computer Graphics]: Fractals

1. Introduction

Interactive rendering of terrain datasets is one of the classical challenges in computer graphics. The rendering of very large terrain geometry is important in a number of application domains, such as scientific visualization, flight simulation, GIS and recent computer games. From a developers point of view terrain rendering is much more than just the real-time display of a landscape [RF02]. In these applications, the terrain rendering is only a stage of a pipeline where the overall resources must be balance accurately. In particular, it is only one task among many others that have to be carried out for each frame as for instance dynamic lighting, dynamic shadowing, and particle system and so on.

Therefore, such applications need an accurate visualization of large terrain datasets at high frame rates without exceeding available graphics resources. Levels of detail based approaches have been extensively used for interactive terrain rendering. In these approaches, the CPU extracts the appropriate levels of detail in a view-dependent manner and the geometry extract are sent to the graphics hardware for rendering at each frame. If the CPU is not capable to efficiently extracting the geometry from the datasets or the communication between the CPU and the GPU is bottlenecked the result is an unacceptable low frame rate.

Current graphics hardware provides common features for both vertex and fragment processors. These features are useful for generating various effects such as displacement mapping. These mapping algorithms take sample points and displace them perpendicularly to the normal of the macrostructure surface with the distance obtained from the height map [SKU08]. In per-vertex displacement mapping the sample points are the vertices of a tessellated mesh.

In this paper we present a novel framework for interactive terrain rendering (Figure 1). At each frame, our algorithm selects a set of active meshes in a view-dependent manner from a rings structure. For each active mesh the fragment processor decodes the associated height map which has been off-line encoded by a fractal compressor. The sampling is performed by vertex processor fetching the elevation components from the height map. In addition, since it is more

convenient to keep large terrain data in a hard drive we provided an out-of-core support to efficiently fetch and access height maps through a caching mechanism.

Our approach provides the following advantages:

- **Efficiency**. The meshes are created in such a way that the graphics hardware can process it quickly. Each vertex is computed independently of the other vertices using only a vertex shader within the inherently parallel GPU. This leads to efficient interactive terrain rendering on GPU.
- **Fully GPU based**. Most of the computations are performed by the GPU. Our approach implies using as little CPU processing power as possible. In real life application, such as computer gaming, this advantage would be very valuable because, it frees the CPU to focus on physics, AI, voice-over-ip, networking, etc...
- **Seamless transition**. During rendering each vertex is informed about its neighbors' meshes. In this way, the seamless transition between neighboring meshes with different resolution is achieved by stitching edge vertices with the finer tessellation level.
- **Fast culling**. The simple data structures used to recover active patches from viewer position allows fast CPU localization of the geometry inside the view frustum.
- **Compression**. The terrain is stored as compressed tiles using fractal compressor. The recursive nature of terrain data fits well with the capabilities of fractal compression allowing good image quality at low bit-rates. Besides, fractal image compression offers interesting features like fast decoding and independent-resolution which are very useful for real time rendering applications.
- **Simplicity**. There are no complex data structures to implement. Our algorithm promotes use of the GPU-compatible data structures such as vertex buffer objects and textures.

In the rest of this paper, we first overview the related work in the area of terrain rendering. Then, we present each component of our algorithm followed by implementations details and results. Finally, we conclude this work and outline our plans for future work.

2. Related Work

In this section we overview related work in level of detail terrain rendering based on CPU and GPU.

In the CPU based approach, the investigation of multiresolution methods to dynamically adapt render model complexity is a very active computer graphics research area as reported in the survey of Pajarola et al. [PG07]. In these algorithms CPU selects the appropriate geometry which is sent to the graphics hardware at each frame. To reduce CPU load several approaches partition the terrain into patches at different resolutions. In order to reduce communication between the CPU and graphics hardware several algorithms utilize geometry cache.

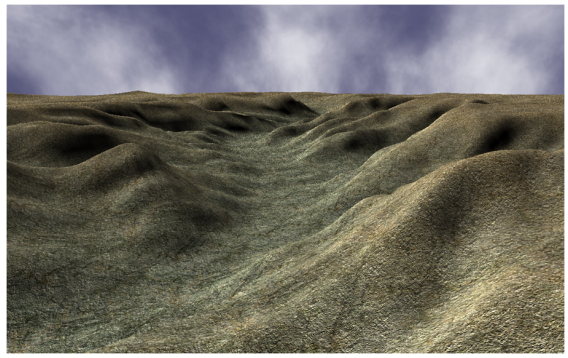

Figure 1: *View of the Grand Canyon rendered by our algorithm.*

The advances in graphics hardware programmability allow developers to leverage GPU processing power for algorithms that operate on terrain rendering. Geomorphes to render terrain patches of different resolution have been used in [Dan03, HDJ04]. In [LH04] the authors present a terrain rendering algorithm based on clipmaps. The clipmap focus follows the position of the viewer. Therefore the area near the viewer can be rendered at high levels of detail while the regions further away are displayed in a lower resolution. In [AH05] the authors improved the performance of this method by moving nearly all rendering operations to the GPU, leaving only decompression and clipmap updating to the CPU.

A persistent grip mapping which covers the entire screen has been used in [LSGES08]. Using perspective mapping the GPU maps each vertex of the perspective grid onto the terrain in such a way that the visible region is remeshed in a view-dependent manner with local adaptivity.

In [LKES07] subdivide the terrain into rectangular patches at different resolution. Each patch is represented by four triangular tiles which are stitched using four tiles in a seamless manner. Here, the different approach is that resolution changes not across patches but within patches. The GPU generates the meshes of the patches by using scaled instances of cached tiles assigning elevation for each vertex from the cached texture.

3. Our approach

We present an algorithm for interactive terrain rendering that fully exploits the current graphics features, such as programmable vertex, displacement mapping and fragment processors.

Our algorithm involves two preprocessing stages. The first is an off-line stage in which the terrain is partitioned into rectangular tiles and each tile is encoded by a fractal compressor. The second stage occurs just before entering in the

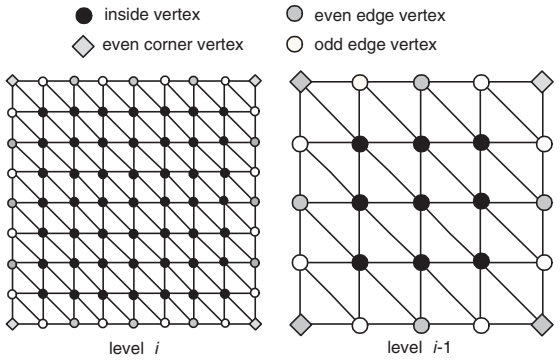

Figure 2: *A patch at level i and a successive patch at level $i-1$ with a schematic representation of vertices.*

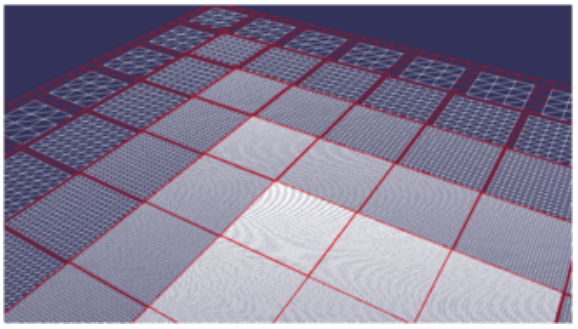

Figure 3: *The rings structure. The center is occupied by the viewer and has a patch with maximum LOD. Around the center the rings at different level of details with a more coarse-grained resolution as the rings move away from the center.*

rendering loop. It generates several planar meshes, which we called patches, at different resolutions without elevation and color components. These patches are stored in the main memory and will be used for displacement mapping in the vertex shader during rendering.

At runtime the visible tiles are selected from a grid structure based on view parameters. Whether it is required, the fragment processor operates a real-time decompression through a progressive refinement in order to obtain the height maps from tiles with a quality proportional to distance from the viewer. In the rendering phase, a vertex shader performs displacement mapping based on elevation components sampled from a height map taken as input.

3.1. Construction of patches

The geometry patches upon which to perform the displacement mapping is a set of static meshes with different levels of detail. Each level contains an $n \times n$ array of vertices stored as a vertex buffer in video memory. As illustrated in Figure 2, in the level $i-1$ the patch contains an array of vertices which is one-quarter of level i.

To permit an efficient and simple way to stitch adjacent edges of different patch resolution each vertex has a 4-tuple (x, y_{pos}, z, w_{odd}) record. The (x,z) is the vertex coordinate, y_{pos} is a flag used to mark its position that is, whether it is a inside vertex, an edge vertex (up, right, down, left) or it is a corner vertex and w_{odd} is a flag used to indicate an odd or even position (see Figure 2). The w_{odd} attribute assures that for any couple of adjacent vertices v_0 and v_1 on an edge, v_0 is in an odd position and v_1 is in an even position or viceversa.

3.2. Rings structure

Using an approach similar to [LH04] we arrange the terrain as a sequence of concentric rings at different resolutions and centered on the viewer's position as illustrated in Figure 3.

In the viewer position there is a central patch with maximum resolution. For each level i of the structure, we define a ring as a set of patches with a level of detail i (see Figure 4). In this way, recursively as rings move away from the viewer they have a progressive coarse-grained resolution.

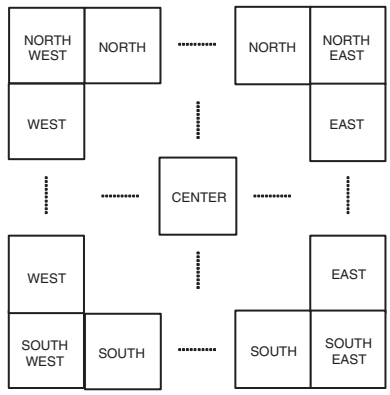

Figure 4: *The rings structure with the center patch and a generic ring at level i. The set of patches at level i are oriented with respect to the position of the center patch.*

During the viewer motion as the desired active rings detect, the patches inside the rings should also update accordingly. In particular, when the viewer moves from the central patch to one of the eight adjacent patches it is selected as the new central patch. Note that the central patch and patches of the first ring have the same level of detail which ensures a seamless transition from the central patch to one of the adjacent patches.

The rings structure is a simple bidimensional array stored in the main memory. It is accessed toroidally through a mod operation which allows a 2D wraparound addressing to permit efficient incremental updates. The rings structure has a

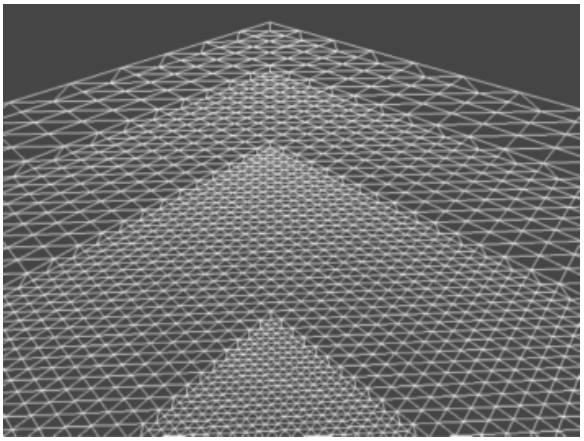

Figure 5: *The patch grid. Note the different resolutions of each ring and how the cracks are resolved between two adjacent patches.*

function $updateRings(i, j)$. This function takes a new central patch (i, j) as input and updates the structure creating the concentric rings and different resolutions. Furthermore, the orientation of each patch is marked with respect to the new central patch (see Figure 4).

3.3. Rendering

At runtime, the rings structure is used to guide the selection of the various concentric rings with correspondent levels of detail based on view-parameters. For each frame, in order to perform the displacement mapping using vertex shader the CPU produces a stream of active patches. Then, we apply view frustum culling as follows. For each patch, we maintain the minimum and maximum elevations values for the local height map. Each patch is extruded to form an axis-aligned bound box which is intersected by the frustum with an optimized view frustum culling described in [AM00].

Before geometry of active patches is streamed to the graphics hardware, the GPU decodes the associated tiles whether it did not decode previously in the video memory. A fragment shader decodes a tile as textures in a finite number of steps n, where n could be selected based on the level of detail of bounded patch (Sec. 4).

During the displacement mapping vertex shader must take into account that two adjacent patches could be disagree on the resolution of the common edges. Then, it is needed to stitch adjacent vertices avoiding cracks and degenerate triangles. To prevent possible cracks in the polygon representation, every patch edge is stitched with respect to the resolution of its adjacent regions. In particular, given two patches with different levels of detail, the triangularization of the edge with a level of detail i is rearranged with respect of the edge with a level of detail $i-1$ as illustrated in Figure 5.

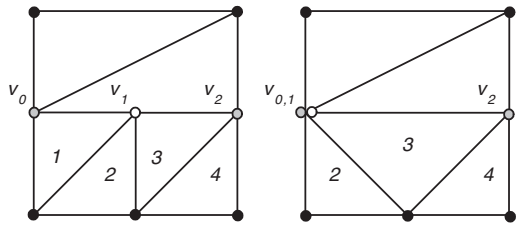

Figure 6: *Stitching patch of level of detail i to $i-1$. Only odd position vertices are rearranged while even position vertices maintain their original position. The elevation values are fetched from adjacent height map for both cases. The triangle 1 turns into a degenerate triangle and will be discarded during rendering.*

The technique used to rearrange vertices on an edge is based on where every single vertex is placed on the edge and on an orientation of its patch as illustrated in Figure 6. We rearrange a vertex using vertex attributes defined in 3.1 as follows. A vertex v_1 is placed in the same position of vertex v_0 if v_1 is an odd edge vertex and v_0 is a previous even edge vertex. Moreover, the elevation values for all edge vertices are fetched from the height map of level of detail $i-1$. The Table below summarizes the cases in which an odd vertex edge must be rearranged based on its position and its patch orientation.

Patch Orientation	odd edge vertex position			
	up	right	down	left
NORTH	×			
EAST		×		
SOUTH			×	
WEST				×

All cases can be efficiently implemented using a look-up texture inside vertex shader. In this way, all vertices are processed independently for each other and in the same way.

4. Fractal Compression

Fractal methods are quite popular in the modeling of natural phenomena in computer graphics as random fractal models of terrain. The fractal compression is defined as the inverse problem that is, given an image translated it in a simple formula. Despite fractal compression has never achieved widespread diffusion it has been proved that it is very effective to achieve very high compression ratios while still maintaining reasonable image quality [Kom97]. Moreover, it offers a sophisticated form of interpolation sometimes referred to as resolution enhancement and very fast encoding phase.

The basic idea of fractal compression is to find similarities between larger and smaller portions of an image. This is accomplished partitioning the original image into blocks of

fixed size, called *range* and creating a shape codebook from the original image of double size of the range, called *domain*. Range blocks partition the image so that every pixel is included while the domain blocks can be overlapped and/or to not contain every pixel. Below we give an overview of fractal compression, the mathematical theory about these principles can be found on [Yuv94].

Encoding. Given a range block R we must find a domain D from codebook such that $R \approx sD + o\mathbf{1}$ where s and o are called *scaling* and *offset* respectively. These values define the optimal transformation by which we can encode an image portion using another part. The encoder must scan all the codebook to find optimal D, s, and o. The domain block must be shrunk by pixel averaging to match the size of range block.

The method of least squares to find the optimal coefficients can be used. Given the two blocks R and D with n pixel intensities, $r_1,\ldots,r_n$ and $d_1,\ldots,d_n$, the quantity to minimize is $\sum_{i=1}^{n}(s\cdot d_i + o - r_i)^2$ where coefficients s and o are given by

$$s = \frac{n\left(\sum_{i=1}^{n} d_i r_i\right) - \left(\sum_{i=1}^{n} d_i\right)\left(\sum_{i=1}^{n} r_i\right)}{n\sum_{i=1}^{n} d_i^2 - \left(\sum_{i=1}^{n} d_i\right)^2}$$

$$o = \frac{1}{n}\left(\sum_{i=1}^{n} r_i - s\sum_{i=1}^{n} d_i\right)$$

In our work, the values s, o, and the position of the domain block D are encoded and stored in a texture.

Decoding. The output encoder is a description of an operator which serves as approximation of the original image. An operator T is defined over an image f as $Tf \equiv sf + o\mathbf{1}$. Thus, starting from any initial image f_0 and applying interactively F to obtain

$$f_1 = Tf_0,\quad f_2 = Tf_1,\quad f_3 = Tf_2,\ldots$$

the sequence f_i converges to an approximation f_n of the original image called attractor after few iterations.

Such decoding phase is very simple and it can be performed using a very efficient fragment program. More precisely, we perform an image rendering to decode each pixel p_{i+1} of image f_{i+1}. For each one we fetch scaling factors s and o from a texture and its code block position which is used to fetch pixels from f_i and perform the following operation:

$$p_{i+1} = p_i \cdot s + o$$

Through experimentation, we have found that the height map assigned to the patch with maximum resolution after 8 interactions converges to the original tile. For patches with a lower level of detail the encoding phase can be performed with less interaction.

5. Implementations and Results

In our current implementation to handle large datasets we have implemented an out-of-core support based on caching. This scheme stores the height maps in disk storage and cache in the video memory only the portions necessary for rendering. When the rendering stage needs a compressed tile, it sends a request to tile memory manager which first checks to see if it is resident in the video memory. If it is, an instance is returned to the rendering stage. If the height map is not resident in the video memory then it is loaded from file system and placed in the video memory. For each entry in the cache we have two attributes. A timestamp used to record last access time and the level of decompression of the height map. When a cache miss occurs the tile manager adopts a strategy based on last recently used as replacement algorithm to select a victim.

We use a $16K \times 16K$ grid of the Grand Canyon area as our main terrain dataset. The terrain dataset has been divided in 4096 tiles of 256×256 pixels and is compressed from 256MB to 96MB that is 24KB for each tile using a range block of 4×4 size. The mesh with maximum resolution has 128×128 vertices which involves a ratio of 4 pixel per vertex. The experiments have showed that these values did not produce any visible artifacts. We have tested our implementation on an AMD 3000 with 1GB memory, and nVidia GeForce 7800 GTX graphics card with 512MB texture memory. The rendering resolution is 1024×768.

The performances are measured using a predefined walkthrough of the camera around the scene. In Table 1 we summarized the performance in case the tiles are decompressed entirely to maximum detail. The frame rates depend mainly on the number of triangles. The first row shows 490 FPS for the base approach with view frustum culling for about $220K$ triangles on average and a cache size of 40 entry, and the third row reports 161 FPS with view frustum culling but all the patches have the same level of detail for about 1M triangles on average. Such observation reveals that selection of active patches is negligible with respect to the total rendering time. Note that without frustum culling the performance fall down rapidly due to the increasing number of triangles as well as numerous cache misses.

The last row shows 521 FPS for an approach based on the incremental fractal decoder particularity. For each tile we use the level of detail of its patch and so the distance from the viewer to determine the number of iterations in the decoding phase. In such a way, the tiles farther from the viewer are decoded with less interaction and they are further decoded as the viewer approaches more closely. In this case, it is clear that we trade the quality required by the lower level of detail patch with a very efficient decompression. From our experiments this benefit produces high quality seamless rendering without visible artifacts with a higher frame per second.

In order to compare the results of our algorithm with other know terrain rendering algorithms on common base we use

Approach	FPS Min	FPS Max	FPS Med	Cache Miss
Base	418	563	490	558
Base Without Frustum	6	8	6.9	23773
Brute Force	161	198	174	558
Brute Force Without Frustum	5	7	6.1	23773
Base Incremental	471	599	521	558

Table 1: *Runtime performance. In first fours approach the decompression is performed entirely. In the last approach the decompression is progressive.*

the expected performance estimated in [LKES07]. In this work, the authors achieve 53M textured triangles per second on average with our same hardware while on comparable hardware expect that BDAM [CGG*03], Clipmap [AH05], and the approach used in [HDJ04] will achieve about 46M, 44M and 43M textured triangles per second, respectively. Our algorithm manages to achieve 64M on average.

Figure 7 shows the textured and wire-frame representation of a terrain generated from the Grand Canyon terrain dataset using our algorithm.

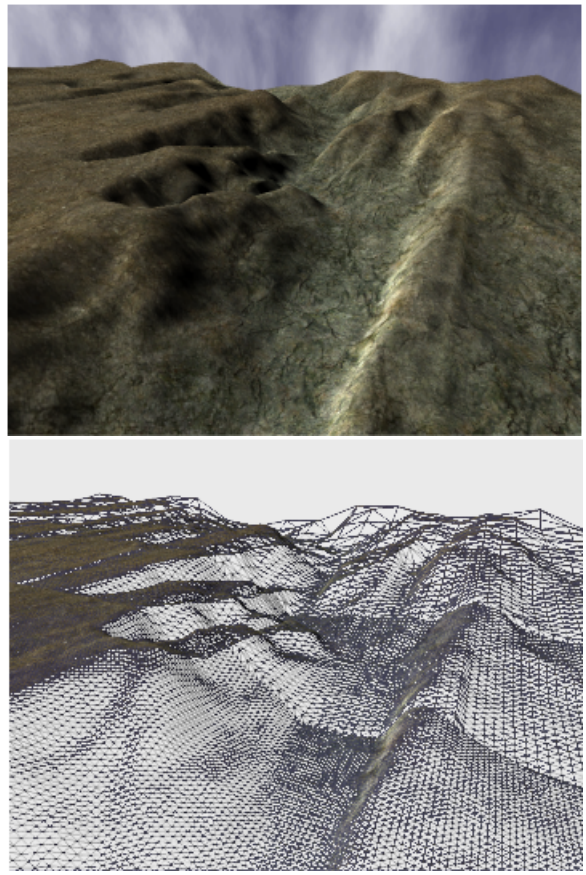

Figure 7: *A terrain view (up) and its wire-frame representation (down)*

6. Conclusion and Future Work

As has been pointed out in [Dan03], today computer graphics applications such as games require for an attractive terrain rendering about 10000 triangles which must be drawn using as little CPU processing power as possible. In fact, in real life application the CPU usually has more things to do than just drawing terrain.

In this work, we have presented a novel approach for real time large terrain rendering by utilizing advanced features of current graphics hardware. The simple rings data structure allows to reduce the CPU load and reduces communication between the CPU and the GPU. Fractal height maps handling offers significant saving storage and real time decompression. These advantages fit well in applications where the graphic resources must be balanced carefully with other real-time graphics techniques.

Our current works include the geometry synthesis within a geometry shader and fractal image compression for the terrain textures. In the future, we are going to investigate a client-server architecture for supporting interactive high quality remote visualization of large terrain.

References

[AH05] ASIRVATHAM A. P., HOPPE H.: Terrain rendering using gpu-based geometry clipmaps. In *GPU Gems II* (2005), Addison-Wesley, pp. 27–44.

[AM00] ASSARSSON U., M LLER T.: Optimized view frustum culling algorithms for bounding boxes. *Journal of Graphics Tools 5*, 1 (2000), 9–22.

[CGG*03] CIGNONI P., GANOVELLI F., GOBBETTI E., MARTON F., PONCHIO F., SCOPIGNO R.: Bdam - batched dynamic adaptive meshes for high performance terrain visualization. *Computer Graphics Forum 22*, 3 (Sept. 2003), 505–514.

[Dan03] DANIEL W.: Terrain geomorphing in the vertex shader. In *ShaderX*^{2}*:Shader Programming Tips & Tricks with DirectX 9* (2003), Wordware Publishing.

[HDJ04] HWA L. M., DUCHAINEAU M. A., JOY K. I.:

Adaptive 4-8 texture hierarchies. In *VIS '04: Proceedings of the conference on Visualization '04* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 219–226.

[Kom97] KOMINEK J.: Advances in fractal compression for multimedia applications. *Multimedia Syst. 5*, 4 (1997), 255–270.

[LH04] LOSASSO F., HOPPE H.: Geometry clipmaps: terrain rendering using nested regular grids. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers* (New York, NY, USA, 2004), ACM, pp. 769–776.

[LKES07] LIVNY Y., KOGAN Z., EL-SANA J.: Seamless patches for gpu-based terrain rendering. In *15th WSCG 2007* (University of West Bohemia, Univerzitni 8, Box 314, CZ 306 14 Plzen, Czech Republic, Jan. 2007), Skala V., (Ed.), WSCG 2007 Full Papers Proceedings, WSCG, University of West Bohemia. Full Paper.

[LSGES08] LIVNY Y., SOKOLOVSKY N., GRINSHPOUN T., EL-SANA J.: A gpu persistent grid mapping for terrain rendering. *Vis. Comput. 24*, 2 (2008), 139–153.

[PG07] PAJAROLA R., GOBBETTI E.: Survey on semi-regular multiresolution models for interactive terrain rendering. *The Visual Computer 23*, 8 (2007), 583–605.

[RF02] ROETTGER S., FRICK I.: The Terrain Rendering Pipeline. In *Proc. EWV '02* (2002), OCG Schriftenreihe, R. Oldenburg, Vienna, pp. 195–199.

[SKU08] SZIRMAY-KALOS L., UMENHOFFER T.: Displacement mapping on the gpu - state of the art. *Computer Graphics Forum 27*, 1 (Jan. 2008).

[Yuv94] YUVAL F.: *Fractal Image Compression - Theory and Application*. Springer-Verlag, New York, 1994.