Visual Comput (2009) 25: 197–208
DOI 10.1007/s00371-008-0214-3

ORIGINAL ARTICLE

Yotam Livny
Zvi Kogan
Jihad El-Sana

Seamless patches for GPU-based terrain rendering

Published online: 11 March 2008
© Springer-Verlag 2008

Y. Livny (✉) · Z. Kogan · J. El-Sana
Ben-Gurion University of the Negev,
Beer-Sheva, Israel
{livnyy, koganz, el-sana}@cs.bgu.ac.il

Abstract In this paper we present a novel approach for interactive rendering of large terrain datasets. Our approach is based on subdividing a terrain into rectangular patches at different resolutions. Each patch is represented by four triangular tiles that are selected form different resolutions, and four strips which are used to stitch the four tiles in a seamless manner. Such a scheme maintains resolution changes within patches through the stitching strips, and not across patches. At runtime, these patches are used to construct a level-of-detail representation of the input terrain based on view-parameters. A selected level of detail only includes the layout of the patches and their boundary edges resolutions. The layout includes the location and dimension of each patch. Within the graphics hardware, the GPU generates the meshes of the patches by using scaled instances of cached tiles and assigns elevation for each vertex from cached textures. Since adjacent rectangular patches agree on the resolution of the common edges, the resulted mesh does not include cracks or degenerate triangles. Our algorithm manages to achieve quality images at high frame rates while providing seamless transition between different levels of detail.

Keywords Terrain visualization · View-dependent rendering · Level-of-detail rendering · Hardware acceleration

1 Introduction

Interactive visualization of landscapes and outdoor environments is important for various graphics applications, such as computer games, flight simulators, and virtual exploration of remote planets. Terrain and height field datasets are vital components of these virtual environments.

The advances in satellite imaging and cartography technologies have led to the generation of large terrain datasets that contain billions of samples. Such terrains usually exceed the rendering capability of currently available graphics hardware, and thus reducing their complexity is mandatory for interactivity. Adjusting the terrain triangulation in a view-dependent manner is a common approach for interactive terrain rendering. Adaptive level-of-detail rendering not only simplifies the geometry, but also manages to reduce aliasing artifacts that may result from rendering uniform dense triangulation.

The challenges of interactive terrain rendering have attracted the interest of researchers for several decades and interesting approaches have been developed (see Sect. 2). Classic level-of-detail rendering schemes generate, usually off-line, multiresolution hierarchies. These hierarchies are used at runtime to guide the selection of appropriate levels of detail based on view-parameters. Some of

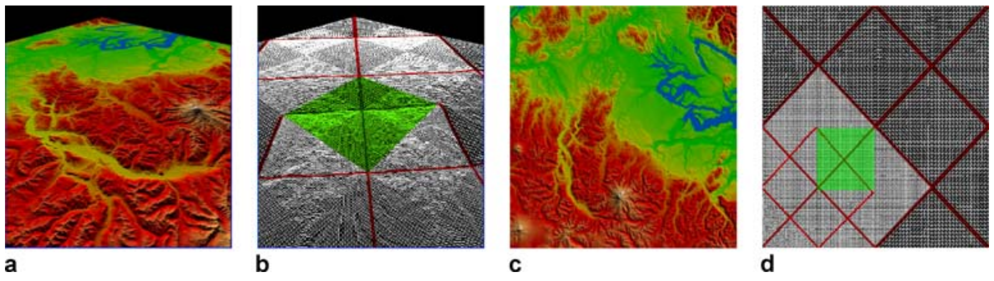

Fig. 1a–d. Terrain rendering using seamless patches. **a** A selected view. **b** The wire-frame of **a**, where the *green* region marks one patch. **c** Top view of **a**. **d** The wire-frame of **c** with the same marked patch

these approaches utilize temporal coherence among consecutive frames while others generate the geometry for each frame, independent of its previous frames. These approaches were able to accelerate the rendering of large terrains, but they were not able to maintain the improvement rate as the speed of the graphics hardware grow faster. The generation of a frame's geometry is performed by executing refine and simplify operations on the CPU, which often fails to complete these computations within the duration of one frame. In addition, the geometry of a frame, which is transferred to the graphics hardware in each frame, often exceeds the bandwidth of the communication channel and results in unacceptably low frame rates.

To reduce the computation load on the busy CPU, several approaches partition the terrain into patches at different resolutions. At runtime, these patches are stitched together to generate a level-of-detail representation, which is then transmitted to the graphics hardware. Stitching these patches in a seamless manner is the main challenge for these approaches. Introducing degenerate triangles and dependencies among patches are used to handle these problems. However, these solutions may introduce visual artifacts, or require additional random-access memory references.

In this paper, we present a novel approach for interactive rendering of large terrain datasets, which is designed to prevent the mentioned limitations of previous algorithms. Our approach subdivides the terrain into rectangular patches at different resolutions, as shown in Fig. 1. Each patch is represented by four triangular tiles that are selected from a small set of predetermined discrete resolutions. The triangular tiles are stitched together by four strips, as shown in Fig. 2. Since the number of different resolutions is very small, the number of required patterns of stitching strips is also very small.

At runtime, the generated patches are used to construct a view-dependent level-of-detail representation. The selected levels of detail do not include any geometry; instead they only include the layout of the patches and the resolutions along their boundary edges. The layout includes the location and dimension of each patch. The resolutions along boundary edges are used to guide the selection of the adequate tiles and strips, for each patch, without the need to query its adjacent patches. Since the resolution is computed per edge, adjacent patches have the same resolution along their shared edge and, as a result, the generated mesh does not include any cracks or degenerate triangles. For each patch p the graphics hardware selects the appropriate templates of cached tiles and stitching strips based on the resolution assigned for the boundary edges of p. These tiles and strips are then transformed (scaled, translated, and/or rotated) to match the location and dimension of the patch p. Finally, the vertex and fragment processors assign elevation and color components for each vertex using the cached textures. To handle large terrains, we provide external texture memory support that caches the necessary displacement and color maps in video memory.

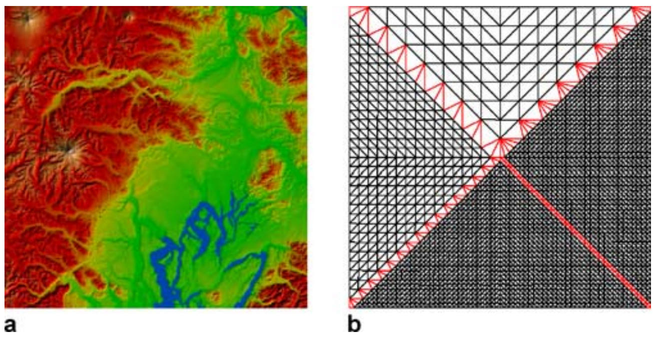

Fig. 2. **a** An image of one patch. **b** The wire-frame of the same patch which shows the stitching of different resolution tiles

Our approach provides several advantages over previous terrain rendering schemes:

– Stitching the boundary of adjacent terrain patches is the main challenge for the patch-based approaches. Previous stitching techniques include monitoring the boundary when the parent patch splits into its children patches, forcing a patch to simplify or refine to comply with the resolution of its adjacent patch, or adding degenerate or sliver triangles to fill the crack between adjacent patches. These approaches maintain random-access memory references between adjacent patches.

In contrast, our algorithm determines the level of detail of each patch without querying its adjacent patches.
- The rendered mesh does not include any degenerate or sliver triangles, since our approach ensures the same triangulation on the two sides of each boundary edge.
- Typical view-dependent rendering schemes require large memory to store the patch hierarchies and their geometry. In our algorithm the hierarchy stores only the patch layout, which includes the location and dimension of each patch, and their geometry as displacement maps. For terrain datasets the layout structure is inferred from the quadtree (the patch hierarchy). As a result, our approach extracts the layout of patches using an implicit representation.
- The communication overhead is reduced as a result of transmitting only the layout of the patches, while using cached tiles and displacement maps to generate level-of-detail representations.

In the rest of this paper we briefly overview related work in terrain rendering. Then we discuss our approach, followed by implementation details and experimental results. Finally, we draw some conclusions and suggest directions for future work.

2 Related work

In this section we briefly overview related work in level-of-detail terrain rendering. We shall focus on approaches that utilize the special properties of terrain datasets.

In general level-of-detail rendering algorithms, terrains are represented as triangulated meshes. Algorithms of this type usually utilize temporal coherence and achieve the best approximation of the terrain for the given view-parameters and triangle budget. However, these algorithms are required to maintain mesh adjacency and validate refinement dependencies at each frame. Several approaches use Delaunay triangulation to constrain the mesh [7, 8, 35], while others use less constrained multiresolution hierarchies with arbitrary connectivity [11, 15, 19, 30].

Level-of-detail algorithms for height-field datasets are, usually, based on regular grid representation. They utilize the longest edge bisection scheme to simplify the memory management by using a restricted quadtree triangulation [2, 33], triangle bintrees [14, 25], or hierarchies of right triangles [16, 26]. However, updating the mesh at each frame prevents the use of geometry caching and the utilization of efficient rendering schemes.

To reduce CPU load and utilize efficient rendering, several approaches subdivide the terrain into square patches at different resolutions. At runtime, the appropriate patches are selected, stitched together, and transmitted to the graphics hardware [18, 33, 34]. Cignoni et al. [6] and Yoon et al. [40] have developed similar approaches for general 3D models. The main challenge, for these approaches, is to stitch the boundaries of the appropriate patches seamlessly. In each frame, the generated representation is transmitted, and rendered. These approaches try to maximize the utilization of the graphics hardware rendering capabilities. However, such utilization is limited by the bandwidth of the communication channel.

To overcome the communication bottleneck, several algorithms have utilized cached geometry. Some approaches [4, 5, 22, 24] cache triangulated regions in texture memory, while others exploit the geometric locality to improve the cache efficiency [20].

Terrains usually compensate small geometric details by textures, and as a result are often accompanied by huge texture maps. Tanner et al. [38] introduced the texture clipmaps hierarchy, and Döllner et al. [13] developed a more general texture hierarchy to handle texture maps. For the graphics hardware, these caching techniques enable fast transition of geometry and texture. Since the cache memory is limited, large datasets may still involve communication overhead.

Cook [9] introduced the displacement maps to represent elevation maps as vertex textures. Several approaches used programmable graphics hardware and displacement maps for efficient adaptive rendering of graphics models [12, 17, 28, 31].

The advances in graphics hardware programmability and processing power have driven the development of a new generation of level-of-detail rendering algorithms. Losasso et al. [29] and Bolz and Schröder [3] used the fragment processor to perform mesh subdivision. Southern and Gain [37], and Larsen and Christensen [23] used the vertex processor to interpolate different resolution meshes in a view-dependent manner. Wagner [39] and Hwa et al. [21] used GPU-based geomorphs to render terrain patches of different resolutions. Dachsbacher and Stamminger [10] used GPU programmability to generate and render procedural details for terrains at runtime. Schneider and Westermann [36] suggested progressive transmission to reduce the data transfer between the CPU and the GPU. Livny et al. [27] cached a persistent grid in the video memory, and projected it on the height-field to generate a view-dependent terrain surface reconstruction.

The clipmaps algorithm [1] uses a set of nested rectangular grids centered around the viewpoint. These grids, which are stored within the video memory, represent different levels of details for the terrain at power-of-two resolutions. In each frame, the visible part of the triangulation is sent to the GPU and modified according to elevation and color maps. However, this algorithm does not perform local adaptivity, and the transition between different levels of detail may result in cracks. The cracks problem is resolved by inserting degenerate triangles, but these triangles may generate visual artifacts such as dark pixels, or Z-buffer fights.

3 Our approach

In this section, we present a novel algorithm for interactive terrain rendering. It partitions the terrain into rectangular patches and utilizes advanced features of graphics hardware, such as programmability, displacement mapping, and geometry caching. Our algorithm involves a light preprocessing stage, generating a hierarchy of elevation maps and color textures, and storing them within the main memory. The coexistence of different levels of detail within the same patch enables seamless stitching of adjacent patches. In each frame, our algorithm uses an implicit patch hierarchy to generate a set of appropriate patches on the fly, based on the current view-parameters. The generated patches are stitched together and sent to the graphics hardware, which adds geometry, per vertex elevation and color.

3.1 Patch scheme

Previous terrain rendering algorithms use either triangular or rectangular patches for view-dependent level-of-detail rendering. On the one hand, algorithms that use rectangular patches assign constant resolution over the entire patch and follow the texture rectangular interface. However, they require fine rectangles to resemble the terrain surface, and impose severe difficulties when stitching adjacent patches. On the other hand, the algorithms that use triangular patches enable easier stitching schemes and better shape approximation, though they suffer incompatibility with the texture rectangular interface and complicate texture management. Our patch scheme combines the advantages of the two schemes; it subdivides the terrain into rectangular patches which consist of triangular tiles. These tiles allow different resolutions to coexist within one patch. Such a scheme provides limited local adaptivity and enables the stitching of adjacent patches in a seamless manner.

In our scheme, a patch is arranged as four tessellated triangular regions which are determined by the two diagonals of the rectangular patch, as shown in Fig. 3. We shall refer to these tessellated triangular regions as *triangular tiles* (or simply *tiles*). The four tiles can have different resolutions which are selected from a predefined set of uniform resolutions. One could treat these tiles as discrete levels of detail of the same tile. Within a patch, the triangular tiles are stitched together by using predefined strips (refer to Fig. 4). Since the number of different resolutions for the tiles is usually small, the number of different stitching strips is also small. Three strip types are required to stitch four tiles of two different resolutions.

We have chosen to adopt tile resolutions at consecutive powers of two to comply with the mipmap resolutions and meet the requirement of Claim 1 (see Sect. 3.4 below).

3.2 Patch hierarchy

The patch hierarchy is constructed top-down by subdividing each patch into 2×2 children patches, similar to quadtree. The patch hierarchy does not store any geometry; instead it stores the position and dimension of each patch with respect to the entire terrain. As a result, it easily fits within local memory, even for very large terrains. In practice, there is no need to implement the hierarchy explicitly. The position and dimension of each patch can be directly retrieved from its parent, using simple shift operations. Therefore, in our algorithm we use the bounding rectangle of the terrain and shift operations, as an implicit hierarchy, to generate the location and dimension of each patch on the fly.

3.3 Runtime rendering

Normalized meshes that represent the triangular tiles and their stitching strips (2 tiles and 3 strips for two different resolutions) are generated only once, and cached in texture memory.

At runtime, the patch hierarchy is used to guide the selection of the various levels of detail based on view-

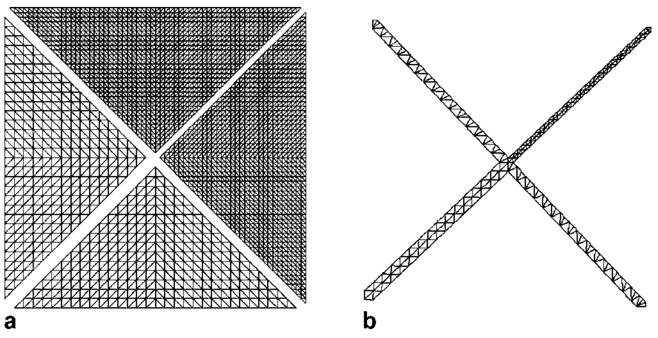

Fig. 3a,b. The components of one patch. **a** The image of four triangular tiles. **b** The image of four strips

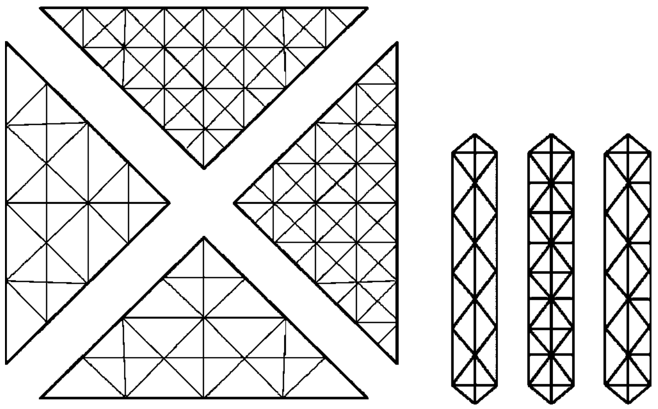

Fig. 4. Triangular tiles at two different resolutions and the required stitching strips

parameters. In each frame the patch hierarchy is traversed in a top-down manner to select a set of active patches that form an appropriate level of detail. The traversal process starts from the root and for each visited patch τ an error value is computed. If the error is too large with respect to the view-parameters, the children of the patch τ are traversed. Otherwise, the resolutions of boundary edges are computed and the patch is added to the list of active patches, which are simultaneously streamed to the graphics hardware for rendering. The resolutions on the boundary edges of the patches are discretized to match that of the triangular tiles. Within the CPU, a patch, p, is represented by its enclosing rectangle and the resolution of its boundary edges, which are enough to determine the tiles and strips required to tile p (see Fig. 5). The resolutions of the four edges of each patch are used to fix the four triangular tiles, then each two adjacent tiles determine the strip that matches their resolution and fills the gap between them. Instances of the cached tiles and strips are transformed to match the enclosing rectangles of the selected patches.

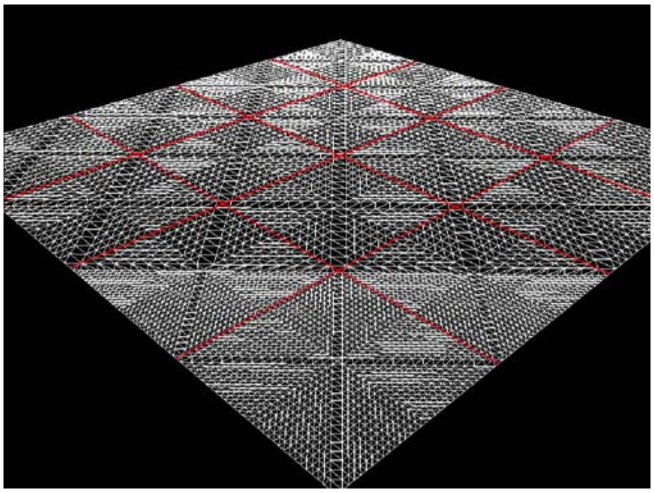
Fig. 5. Stitched tiles and strips of different resolutions. The meshes of triangular tiles and strips appear in *white* color and the patches boundaries appear in *red*

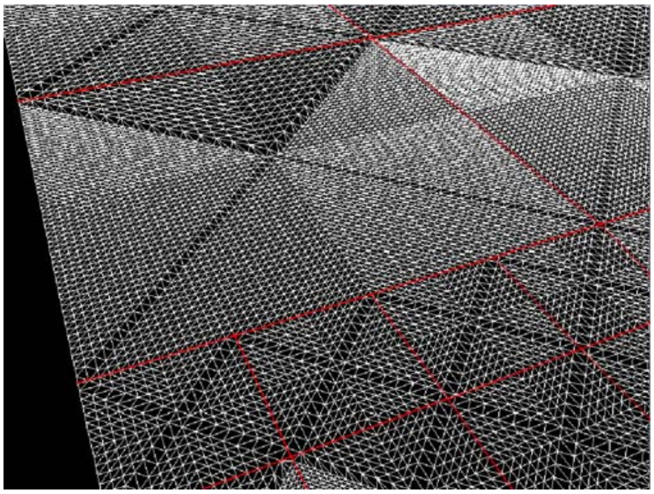
Fig. 6. Surface triangulation extracted from hierarchy

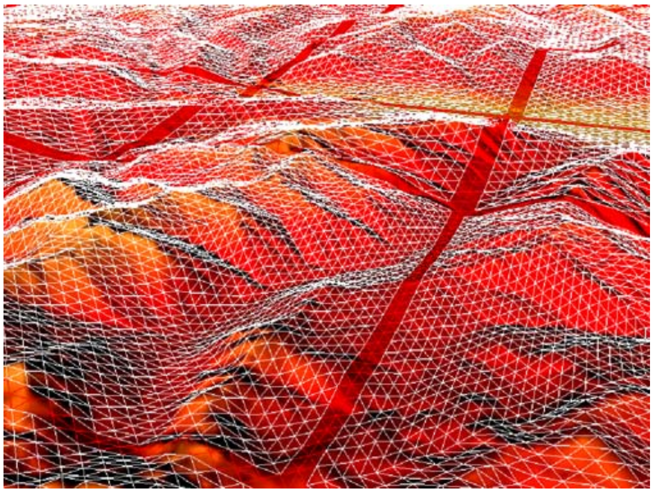
Fig. 7. A terrain view with a wire-frame on top. The meshes of triangular tiles appear in *white* color and the strips appear in *red*

Since we use same computation for adjacent patches, to calculate the resolution on common edges, the stitching of adjacent patches is smooth and does not include cracks or degenerate triangles. The light representation of the active patches contributes to the dramatic reduction on the CPU-GPU communication load.

Tessellating a patch by triangular tiles generates a planar mesh that does not include elevation nor color components (see Fig. 6). These components are assigned for each vertex by the vertex and fragment processors, which use the 2D position of a vertex to fetch and assign the appropriate elevation and color from the cached textures (see Fig. 7).

3.4 Level of detail

The quality of a patch is determined by the resolutions of its tiles, which are determined by the resolution of its boundary edges. The resolutions of an edge is computed based on its length l and its distance d from the viewpoint by using Eq. 1, where ρ is the error tolerance. If ε is larger than 1, the patch is split to its children, otherwise the resolution of the edge is determined by $\varepsilon R_{\max}$ rounded up to the closest resolution, where $R_{\max}$ is the highest available resolution. The scaling factor, which is used to resize a tile to match the patch's enclosing rectangle, also determines the texture level from which the elevation and color values are fetched.

$$\varepsilon = \rho \frac{l}{d} \qquad (1)$$

The transition of the various levels of detail over the surface of the generated mesh is dictated by the tiles' resolutions and patches' levels. Note that in the selected level of detail, adjacent patches can be different at most, by one level in the hierarchy. For that reason, the generated mesh does not include cracks or degenerated triangles as shown in Claim 1.

Claim 1. *The generated terrain triangulation does not include cracks, which means that any two adjacent patches have the same number of triangles on the common edge.*

Proof. Without loss of generality, the proof of the claim is written for two different resolutions and quadtree subdivision. Our proof distinguishes between two cases:

1. The two selected adjacent patches have the same dimension, and thus they share a common edge e along the entire side. By selecting the same tile on the two sides of e, the two patches are stitched seamlessly.
2. The two selected adjacent patches are in different dimensions which means that the edge belongs to one patch on one side and two patches on the other side (see the edge $\overline{AB}$ in Fig. 8). We first show that the tile $\overline{ABJ}$ gets the highest resolution $R2$. The patch $\overline{ABCD}$ has split to four children, which means that one of its edges required a resolution higher than $R2$ (beyond the available resolutions). Let this edge be $\overline{CD}$ and l be the length of the edge $\overline{AB}$, and let the distances of the edges $\overline{AB}$ and $\overline{CD}$ from the current viewpoint be d_{far} and d_{near}, respectively. Based on Eq. 1, $\rho\frac{l}{d_{\text{near}}} > 1$ holds, since the patch $\overline{ABCD}$ has split into its four children, then:

$$\rho\frac{l}{d_{\text{near}}} > 1 \Rightarrow \rho\frac{l}{l+d_{\text{near}}} > \frac{1}{2} \Rightarrow \rho\frac{l}{d_{\text{far}}} > \frac{1}{2};$$

using $d_{\text{far}} \le d_{\text{near}} + l$.
Therefore, the edge $\overline{AB}$ is assigned the resolution $R2$, and the edges $\overline{AE}$ and $\overline{EB}$ are assigned the resolution $R1$. Our algorithm assigns resolution $R1$ to edges with error tolerance ε in the range $[0, 0.5]$ and $R2$ to those with error tolerance in the range $(0.5, 1.0]$.

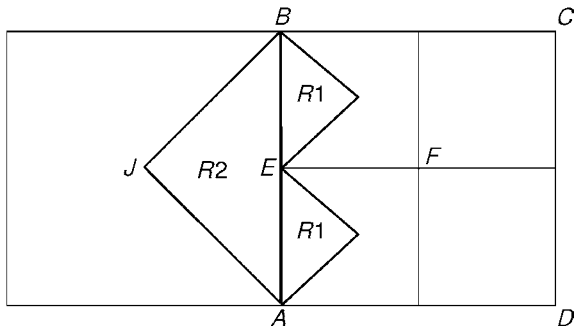

Fig. 8. Stitching tiles at two different levels of detail

3.5 Texture pyramid

Terrain datasets are usually represented by elevation maps and color textures, which store the properties of vertices in the original terrain. We use multiple-level texture pyramids at successive powers of two (similar to mipmaps) to support level-of-detail rendering. These texture pyramids are used at runtime to achieve faithful sampling of the textures for the vertices of each tile. Since these multiple-level pyramids are similar to mipmaps, we could let the hardware construct them. Then at runtime, the vertex processor determines from which level to select the values. However, such an approach does not work when the terrain size exceeds the capacity of the base level of the mipmaps [28].

For large terrains, the multiple-level texture pyramids are constructed once by the CPU before being transferred for caching in the texture memory. We start with the original texture, which represents the most detailed level, and each new level is generated from the previous one by reducing the resolution by half at each dimension. The pixels in the generated level are computed by interpolating the four corresponding pixels of the previous level.

At runtime, an external texture memory manager is used to maintain, within the texture memory, the portions of data necessary for rendering the next frame. The upload from main memory into texture memory is performed by fetching nested clipmaps centered at the viewpoint [1, 38], as shown in Fig. 9. Even though these clipmap levels occupy the same memory size, they cover increasing regions of the terrain (see Fig. 9a, b). Each extracted level is an independent texture that may require separate binding (performed by the CPU) when it is used by a transmitted patch. However, the CPU cannot predict the level of detail of the transmitted patches since they are streamed in an arbitrary order. To overcome this limitation, we apply a technique that uses multiple texture maps within one bounded texture-buffer. It is done by laying all the different texture levels into one large texture-buffer similar to

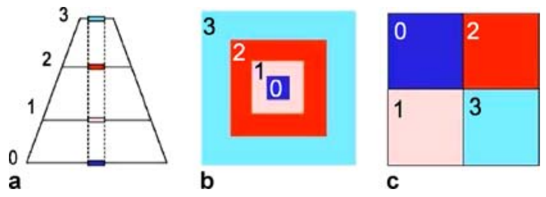

Fig. 9a–c. The layout of nested clipmaps. **a** The texture pyramid stores the entire terrain in decreasing resolutions. **b** Nested clipmaps cover increasing regions of the terrain. **c** Clipmaps are laid on the texture atlas

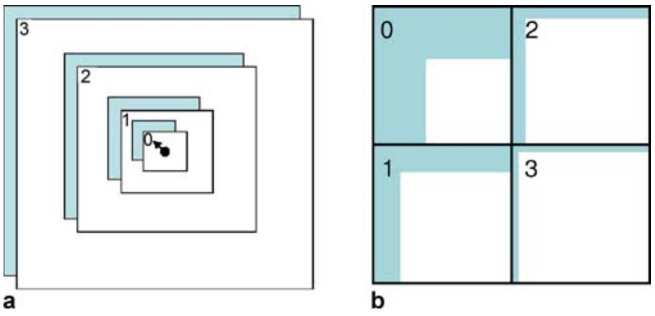

Fig. 10a,b. The modification (*blue* color) in the clipmaps resulting from camera movement. **a** Clipmap layout. **b** F-shape texture atlas

the idea of texture atlases [32]. The uploaded textures tessellate the texture-buffer uniformly because of their equal sizes, as shown in Fig. 9c. In such a scheme, all the levels of detail are accessible from the GPU simultaneously, and there is no need for the CPU to switch textures.

The changes in view-parameters modify the viewed region, usually in a continuous manner. To avoid expensive modification of the entire texture memory and to utilize temporal coherence, each tile undergoes an L-shape modification similar to [1]. A new position for the viewpoint requires the update of the nested clipmaps (in order to keep them centered at the viewpoint). Note that most of the cached data is still relevant and only small portions need to be updated. The update of the texture tiles is simply performed by replacing the irrelevant pixels from the texture with new pixels, and using repeat textures for the tiles. Since the resolution of clipmap levels decreases exponentially, the area that needs to be updated becomes smaller (coarse levels are seldom updated). In general, the updated areas of all the tiles that lay in one texture, form an F-shape (see Fig. 10).

4 Implementation details

We have implemented our algorithm in C++ and Cg for Microsoft Windows. Our implementation uses OpenGL as the graphics API, and requires graphics hardware that supports *nVidia Shader Model 3.0*.

In our current implementation, we do not construct the patch hierarchy explicitly; instead, an implicit representation is used. The root of the hierarchy is the coarsest level of detail that fits in texture memory and matches the interactive rendering capability of the graphics hardware. Hence, the height of the hierarchy can be easily determined based on the hardware capabilities. Note that the 2D bounding rectangle of the root is the same as that of the original terrain. Recall that patch hierarchy does not store any mesh geometry or pixel information. Therefore, the subdivision of a patch into its children is performed by several shift instructions within the CPU. The traversal of the implicit patch hierarchy is performed similar to the explicit one, and often more efficient as a result of avoiding memory access to fetch children patches. According to our experimental results, traversing the patch hierarchy is negligible compared to the total rendering time of a frame (see Sect. 5). View-frustum culling is performed by the CPU during hierarchy traversal that determines the set of active patches. For each patch τ that requires subdivision to reach the appropriate level of detail, the children patches of τ are tested against the view-frustum only if τ intersects the boundary of the view-frustum. If the patch τ is entirely included within the view-frustum, then all its children patches are also within the view-frustum. If τ intersects the view-frustum's boundary, its children patches are tested whether they are inside, outside, or intersect the current view-frustum. The outside-marked patches are culled and not processed further (see Fig. 11).

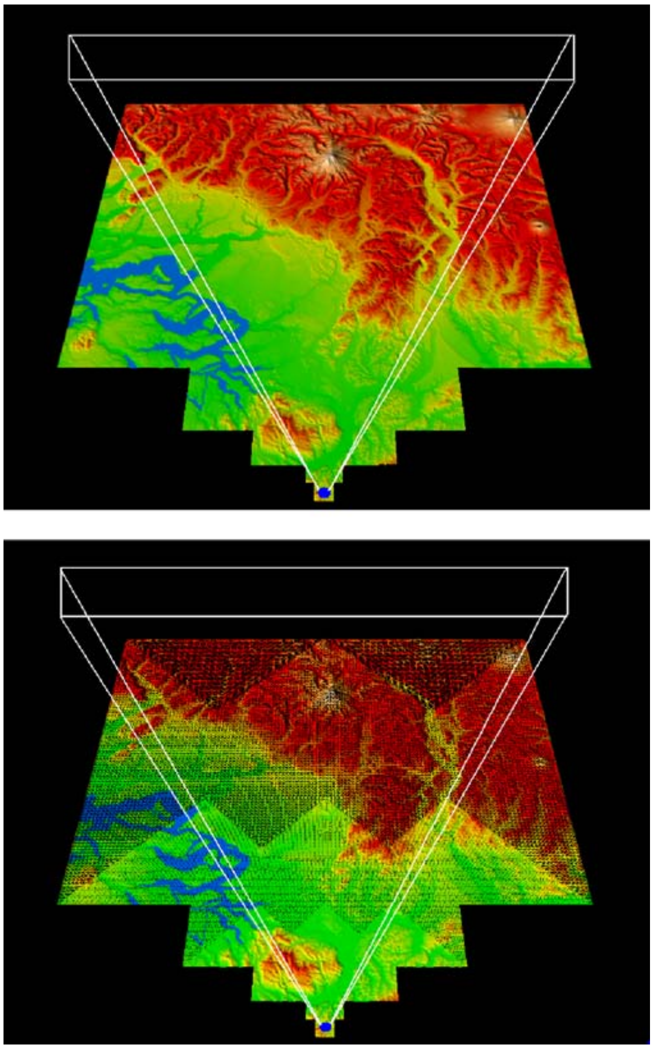

Fig. 11. View-frustum culling. A shaded view (*top*) and its wireframe representation (*bottom*)

The meshes that represent the tiles and strips of different resolutions are cached in texture memory, as mentioned in Sect. 3.3. The meshes are stored in an indexed triangle strip format, and cached using the Vertex Buffer Object (VBO) extension. At runtime, these meshes are used to tile the selected patches. Since the number and the size of these meshes are small (2 tiles and 3 strips are required to support two different resolutions within a patch), we store four orientations of each tile and each strip to avoid rotating and mirroring these meshes at runtime.

To handle large terrain datasets, our algorithm uses an external texture memory support (refer to Sect. 3.5). The implementation of the suggested memory manager uses the Fragment Buffer Object (FBO) extension to support dynamic updates of the textures.

In earlier algorithms, the CPU sends three coordinates for each uncached vertex in the dataset. Our algorithm, utilizes the hardware supported displacement map to pass vertex coordinates to the graphics hardware by sending only elevation value for each vertex. The other two coordinates are generated using the terrain grid structure. This technique reduces the data transfer at runtime from three coordinates to one coordinate for each vertex.

5 Results

Our implementation was tested on an AMD Athlon 3500 with 1 GB memory, and *nVidia* GeForce 7800 GTX graphics card with 256 M texture memory using Puget Sound and Grand Canyon terrain datasets. The terrain datasets we used are of sizes 1.5 GB, 112 MB and 24 MB for 16 K^2, 4 K^2 and 1 K^2 samples, respectively. This section reports and analyzes selected entries of these results.

The performances of our algorithm are summarized in Table 1. For each dataset, we visualize different regions of the terrain to capture the various processing patterns. Each row reports the terrain size, the viewed region, the error tolerance, and the performance with and without view-frustum culling. Two options were recorded for the viewed region: *boundary* and *internal*, which refer to flying near the boundary and inside the terrain, respectively. When the error tolerance equals 1 pixel, we select more detailed levels than when the error tolerance equals 2 pixels. In the performance columns we report the number of rendered triangles (*Triangle* column), the number of traversed patches (*Traversed* column), the number of rendered patches (*Rendered* Column), the number of culled patches (*Culled* column), and the frame rates. The view-frustum culling increases the performances by a factor of 2 when flying on the boundary of the terrain, and by a factor of 3 in general. Our algorithm manages to achieve quality images at high frame rates, as can be seen in Table 1. The frame rates depend mainly on the number of triangles. The first row shows 156 FPS without view-frustum culling for about 330 K triangles and 91 rendering patches, and the sixth row reports the same FPS with view-frustum culling and 56 patches. Therefore, we can conclude that patch selection is negligible with respect to the total rendering time. Note that patch selection also includes view-frustum culling and transmitting the active patches to the graphics hardware.

The intersection of the terrain basis (the XY-plane) with the view-frustum forms a triangular area of visible geometry. Rendering rectangular patches usually results in transmitting many triangles that are invisible (out of view-frustum). Our patch scheme enables rendering only the visible part of a patch, meaning only visible tiles within a single path. Supporting partial rendering for patches improves the rendering speed by about 15% without harming the rendering quality.

The results of our algorithm were compared to the results of three known view-dependent terrain rendering algorithms. We implemented these algorithms based on published papers, while using the same rendering techniques, such as VBO, triangle strips, CG programming, and geometry caching. In order to present reliable comparisons of the performance of these algorithms, we only measured the number of triangles that are actually processed by the GPU. Using our machine configuration the BDAM [4], the clipmap [1], and adaptive 4–8 texture hierarchies [21] render 46 M, 44 M, and 43 M textured triangles per second, respectively. Our algorithm manages

Table 1. Runtime performance

Dataset	View	Error	With frustum culling					Without frustum culling			
size	region	value ρ	Triangles	Traversed	Rendered	Culled	FPS	Triangles	Traversed	Rendered	FPS
4K×4K	Boundary	2	172 696	109	46	36	283	331 428	121	91	156
4K×4K	Internal	2	138 668	109	33	49	380	403 896	141	106	138
16K×16K	Boundary	2	180 731	100	50	27	271	338 240	123	93	153
16K×16K	Internal	2	148 200	126	39	58	354	389 197	151	107	135
4K×4K	Boundary	1	354 248	137	66	37	138	763 864	189	142	69
4K×4K	Internal	1	330 480	133	56	44	156	1 133 796	265	199	52
16K×16K	Boundary	1	422 358	179	82	74	112	915 190	213	160	56
16K×16K	Internal	1	414 966	152	70	63	112	1 359 642	340	231	43

to render 53 M textured triangles per second on average. These numbers show that the simplicity of our GPU shader with the use of the displacement maps provide promising results.

The clipmap algorithm [1] and our approach use the same rendering technique (cached elevation maps, planar geometry transmission, triangle strips, and vertex texturing). To improve our comparisons, we also tested the utilization of the view-frustum culling algorithm. The differences between the number of triangles sent to the GPU and the number of triangles that actually fall in the view-frustum were measured for these two algorithms. For experimental purposes, the number of triangles sent to the GPU was fixed to 500 K. We found that for the clipmap algorithm, the view-frustum includes 380 K triangles, which are only 76% of the transmitted triangles. The partial patch transmission scheme, which prevents the transmission of invisible tiles within a selected patch, improves the culling algorithm utilization. When using the partial patch rendering scheme, the view-frustum in our algorithm includes 440 K triangles, which are 88% of the transmitted triangles. This scheme improves the FPS of our algorithm by 15% over the clipmap algorithm, by providing the same image quality using less triangles.

The contribution of the CPU and the GPU to the performance of the algorithm are shown in Table 2. The first part of each row represents the configuration of the frame, which includes the number of rendered patches (*RP*), culled patches (*CP*), and rendered triangles (*Tris*). The forth and fifth columns report the CPU and the GPU processing times, respectively. The CPU load is tiny and has almost no influence on the frame rates for two main reasons – the selection of the active patches (by the CPU) is very light, and the CPU runs in parallel to the GPU. These contributions are also supported by the results shown in Table 2. In addition, the results show that the bottleneck of our algorithm is the GPU. Therefore, it will benefit from the current trend of improving GPU rates.

Table 2. Hardware performance analysis

Configuration			Time		
RP	CP	Tris	CPU (μs)	GPU (ms)	FPS
18	34	88 986	17.54	1.85	583
22	13	96 246	22.57	1.98	545
27	18	119 054	26.88	2.58	418
39	58	148 200	30.67	3.04	354
50	27	180 731	34.84	3.98	271
47	23	230 372	39.06	5.68	190

Table 3 shows the performance of our suggested external texture memory scheme using various subtexture sizes and three navigation speeds – fast, medium, and slow motion. The subtexture (clipmap) sizes in pixels appear in the first column. The *Entire texture* column reports the time, in milliseconds, for binding the entire texture at each frame. Such a scenario results in unacceptably low frame rates. The *Partial update* columns report the time of the different navigation patterns. The fast navigation implies extensive

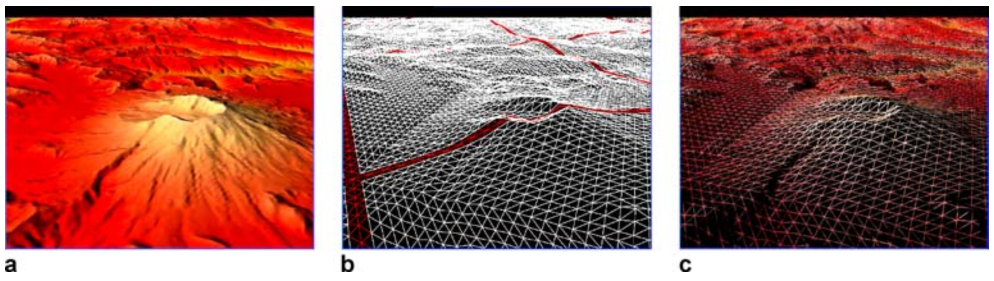
Fig. 12a–c. A terrain view at $\rho = \rho_0$. **a** A shaded surface. **b** Tiles in *white* and strips in *red*. **c** The wire-frame representation

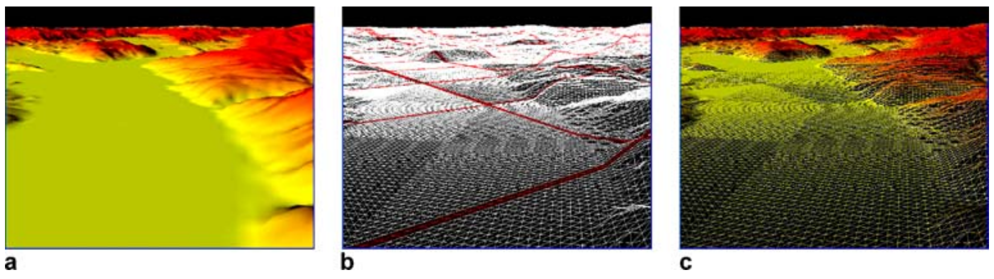
Fig. 13a–c. A terrain view at $\rho = 2\rho_0$. **a** A shaded surface. **b** Tiles in *white* and strips in *red*. **c** The wire-frame representation

Table 3. Out-of-core performances

Clipmap size	Entire texture (ms)	Partial update (ms)		
		Fast	Medium	Slow
128^2	38	1.22	0.17	0.01
256^2	161	1.57	0.63	0.01
512^2	994	3.59	1.39	0.02

updates of the tiles, whereas the slow navigation requires only tiny updates at each frame.

Figure 11 shows the shaded representation above and the wire-frame representation below of a terrain view after applying view-frustum culling, which is performed by the CPU during the selection of active patches.

Figures 12 and 13 were generated from Puget Sound terrain datasets using our algorithm at different error values (ρ). In each figure, the image a shows a shaded view that depicts image quality. Figures 12b and 13b show the wire-frame representation that illustrates the triangular tiles in white color and the stitching strips in red. Figures 12c and 13c show the generated mesh and the transition of the levels of detail.

6 Conclusion and future work

We have presented a novel approach for interactive terrain rendering that reduces the load on the CPU, utilizes texture memory, and leverages advanced features of the GPU. The terrain is subdivided into rectangular patches on the fly. Each patch is represented by four triangular tiles at different resolutions which are stitched together using four triangle strips. At runtime the CPU selects the appropriate patches based on the view-parameters and determines the resolution on their boundaries. The different tiles and stitching strips are cached in texture memory and used to tile each patch according to its boundary resolution. Multiresolution levels of color textures and displacement maps are also cached in texture memory and used by the vertex and fragment processors to assign the elevation and color for each vertex.

Our approach balances the load among the CPU and the GPU and dramatically reduces the communication traffic between them. Adjacent patches are seamlessly stitched without cracks or degenerate triangles, since they have the same resolution on the common edge. Furthermore, each patch determines its own resolution independent of its adjacent patches; it simply selects the different tiles that comply with its boundary resolutions. The use of tiles provides limited local adaptivity which contributes to the smoothness of the generated mesh.

Our algorithm performances are strongly influenced by the number of vertex pipelines. The algorithm relies on the vertex fetch operation which enables the vertex processor to access texture memory. This new feature is not yet optimized, as in the fragment processor. In addition, the current GPUs have a small number of vertex pipelines. We predict that future development on vertex processor hardware will lead to impressive improvements in our algorithm performance.

We see the scope of future work in extending the idea of independent patches to support exact local error schemes. Such development will provide faster view-dependent rendering for large terrain datasets by providing similar image quality, while using fewer triangles. Moreover, our suggested approach generates the patches geometry within the GPU, and hence, cannot utilize temporal coherence among conservative frames. Developing and utilizing temporal coherence within the GPU could contribute to further improvements of the algorithm performance.

Acknowledgement This work is partially supported by the Lynn and William Frankel Center for Computer Sciences and the Tuman fund.

References

1. Asirvatham, A., Hoppe, H.: Terrain rendering using GPU-based geometry clipmaps. In: Pharr, M., Fernando, R. (eds.) GPU Gems 2, pp. 27–45. Addison-Wesley (2005)
2. Bao, X., Pajarola, R., Shafae, M.: Smart: An efficient technique for massive terrain visualization from out-of-core. In: Proceedings of Vision, Modeling and Visualization '04, pp. 413–420 (2004)
3. Bolz, J., Schröder, P.: Evaluation of subdivision surfaces on programmable graphics hardware. Submitted (2005)
4. Cignoni, P., Ganovelli, F., Gobbetti, E., Marton, F., Ponchio, F., Scopigno, R.: BDAM – batched dynamic adaptive meshes for high performance terrain visualization. Comput. Graph. Forum **22**(3), 505–514 (2003)
5. Cignoni, P., Ganovelli, F., Gobbetti, E., Marton, F., Ponchio, F., Scopigno, R.: Planet-sized batched dynamic adaptive meshes (P-BDAM). In: Proceedings of Visualization '03, pp. 147–155. IEEE Computer Society Press, Washington, DC (2003)
6. Cignoni, P., Ganovelli, F., Gobbetti, E., Marton, F., Ponchio, F., Scopigno, R.: Adaptive tetrapuzzles: efficient out-of-core construction and visualization of gigantic multiresolution polygonal models. ACM Trans. Graph. **23**(3), 796–803 (2004). doi:10.1145/1015706.1015802
7. Cignoni, P., Puppo, E., Scopigno, R.: Representation and visualization of terrain surfaces at variable resolution. Visual Comput. **13**(5), 199–217 (1997)
8. Cohen-Or, D., Levanoni, Y.: Temporal continuity of levels of detail in delaunay triangulated terrain. In: Proceedings of Visualization '96, pp. 37–42. IEEE Computer Society Press, Los Alamitos, CA (1996)
9. Cook, R.L.: Shade trees. Comput. Graph. Forum **18**(3), 223–231 (1984)
10. Dachsbacher, C., Stamminger, M.: Rendering procedural terrain by geometry image warping. In: Eurographics Symposium in Geometry Processing, pp. 138–145 (2004)

11. De Floriani, L., Magillo, P., Puppo, E.: Building and traversing a surface at variable resolution. In: Proceedings of Visualization 97, pp. 103–110. IEEE Computer Society Press, Los Alamitos, CA (1997)
12. Doggett, M., Hirche, J.: Adaptive view dependent tessellation of displacement maps. In: HWWS '00: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware, pp. 59–66. ACM Press, New York, NY (2000). doi:10.1145/346876.348220
13. Döllner, J., Baumann, K., Hinrichs, K.: Texturing techniques for terrain visualization. In: Proceedings of Visualization '00, pp. 227–234. IEEE Computer Society Press, Los Alamitos, CA (2000)
14. Duchainear, M., Wolinsky, M., Sigeti, D., Miller, M., Aldrich, C., Mineev-Weinstein, M.: ROAMing terrain: Real-time optimally adapting meshes. In: Proceedings of Visualization '97, pp. 81–88. IEEE Computer Society Press, Los Alamitos, CA (1997)
15. El-Sana, J., Varshney, A.: Generalized view-dependent simplification. Comput. Graph. Forum **18**(3), 83–94 (1999)
16. Evans, W.S., Kirkpatrick, D.G., Townsend, G.: Right-triangulated irregular networks. Algorithmica **30**(2), 264–286 (2001)
17. Gumhold, S., Hüttner, T.: Multiresolution rendering with displacement mapping. In: HWWS '99: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware, pp. 55–66. ACM Press, New York, NY (1999). doi:10.1145/311534.311578
18. Hitchner, L., McGreevy, M.: Methods for user-based reduction of model complexity for virtual planetary exploration. In: SPIE 1913, pp. 622–636 (1993)
19. Hoppe, H.: Smooth view-dependent level-of-detail control and its application to terrain rendering. In: Proceedings of Visualization '98, pp. 35–42. IEEE Computer Society Press, Los Alamitos, CA (1998)
20. Hoppe, H.: Optimization of mesh locality for transparent vertex caching. In: Proceedings of SIGGRAPH '99, pp. 269–276. ACM Press, New York, NY (1999)
21. Hwa, L.M., Duchaineau, M.A., Joy, K.I.: Adaptive 4–8 texture hierarchies. In: Proceedings of Visualization '04, pp. 219–226. IEEE Computer Society Press, Los Alamitos, CA (2004)
22. Lario, R., Pajarola, R., Tirado, F.: Hyper-block-quadtin: Hyper-block quadtree based triangulated irregular networks. In: Proceedings of IASTED VIIP, pp. 733–738 (2003)
23. Larsen, B.S., Christensen, N.J.: Real-time terrain rendering using smooth hardware optimized level of detail. J. WSCG **11**(2), 282–289 (2003)
24. Levenberg, J.: Fast view-dependent level-of-detail rendering using cached geometry. In: Proceedings of Visualization '02, pp. 259–266. IEEE Computer Society Press, Washington, DC (2002)
25. Lindstrom, P., Koller, D., Ribarsky, W., Hodges, L.F., Faust, N., Turner, G.A.: Real-time, continuous level of detail rendering of height fields. In: Proceedings of SIGGRAPH '96, pp. 109–118. ACM Press, New York, NY (1996)
26. Lindstrom, P., Pascucci, V.: Terrain simplification simplified: A general framework for view-dependent out-of-core visualization. IEEE Trans. Visual. Comput. Graphics **8**(3), 239–254 (2002)
27. Livny, Y., Sokolovsky, N., Grinshpoun, T., El-Sana, J.: A GPU persistent grid mapping for terrain rendering. Visual Comput. **24**(2), 139–153 (2008). doi:10.1007/s00371-007-0180-1
28. Losasso, F., Hoppe, H.: Geometry clipmaps: terrain rendering using nested regular grids. ACM Trans. Graph. **23**(3), 769–776 (2004)
29. Losasso, F., Hoppe, H., Schaefer, S., Warren, J.: Smooth geometry images. In: Eurographics/ACM SIGGRAPH Symposium on Geometry Processing, pp. 138–145. Eurographics Association, Aire-la-Ville, Switzerland (2003)
30. Luebke, D., Erikson, C.: View-dependent simplification of arbitrary polygonal environments. In: Proceedings of SIGGRAPH '97, pp. 199–208. ACM Press, New York, NY (1997)
31. Moule, K., McCool, M.D.: Efficient bounded adaptive tessellation of displacement maps. In: Proceedings of Graphics Interface, pp. 171–180 (2002)
32. NVIDIA: Improve batching using texture atlases. SDK White Paper (2004)
33. Pajarola, R.: Large scale terrain visualization using the restricted quadtree triangulation. In: Proceedings of Visualization '98, pp. 19–26. IEEE Computer Society Press, Los Alamitos, CA (1998)
34. Pomeranz, A.: ROAM using triangle clusters (RUSTiC). Master's thesis, U.C. Davis CS Dept. (2000)
35. Rabinovich, B., Gotsman, C.: Visualization of large terrains in resource-limited computing environments. In: Proceedings of Visualization '97, pp. 95–102. IEEE Computer Society Press, Los Alamitos, CA (1997)
36. Schneider, J., Westermann, R.: Gpu-friendly high-quality terrain rendering. J. WSCG **14**(1–3), 49–56 (2006)
37. Southern, R., Gain, J.: Creation and control of real-time continuous level of detail on programmable graphics hardware. Comput. Graph. Forum **22**(1), 35–48 (2003)
38. Tanner, C.C., Migdal, C.J., Jones, M.T.: The clipmap: a virtual mipmap. In: Proceedings of SIGGRAPH '98, pp. 151–158 (1998). doi:10.1145/280814.280855
39. Wagner, D.: Terrain geomorphing in the vertex shader. ShaderX2: Shader Programming Tips & Tricks with DirectX 9 (2004)
40. Yoon, S.E., Salomon, B., Gayle, R.: Quick-VDR: Out-of-core view-dependent rendering of gigantic models. IEEE Trans. Visual. Comput. Graph. **11**(4), 369–382 (2005). doi:10.1109/TVCG.2005.64

YOTAM LIVNY is a Ph.D. student of Computer Science at the Ben-Gurion University of the Negev, Israel. His research interests include interactive rendering of large 3D graphic models using multiresolution hierarchies and the programmable graphics hardware. Yotam received a B.Sc. in Mathematics and Computer Science from the Ben-Gurion University of the Negev, Israel in 2003. He is a Ph.D. candidate.

ZVI KOGAN received a M.Sc. in Computer Science from Ben-Gurion University of the Negev, Israel in 2006, under the advisory of Dr. Jihad El-Sana. His research interest includes interactive rendering of large terrain datasets, and programmable graphics hardware. Zvi received a B.A. in Computer Science from Haifa University, Israel in 1997.

JIHAD EL-SANA is a Senior Lecturer of Computer Science at Ben-Gurion University of the Negev, Israel. El-Sana's research interests include 3D interactive graphics, multiresolution hierarchies, geometric modeling, computational geometry, virtual environments, and distributed and scientific visualization. His research focuses on polygonal simplification, occlusion culling, accelerating rendering, remote/distributed visualization, and exploring the applications of virtual reality in engineering, science, and medicine. El-Sana received a B.Sc. and M.Sc. in Computer Science from Ben-Gurion University of the Negev, Israel in 1991 and 1993. He received a Ph.D. in Computer Science from the State University of New York at Stony Brook in 1999. El-Sana has published over 40 papers in international conferences and journals. He is a member of Eurographics, and IEEE.