Persistent Grid Mapping: A GPU-based Framework for Interactive Terrain Rendering

Yotam Livny Neta Sokolovsky Tal Grinshpoun Jihad El-Sana

Department of Computer Science,
Ben-Gurion University of the Negev,
Beer-Sheva, Israel
{livnyy,netaso,grinshpo,el-sana}@cs.bgu.ac.il

Abstract

In this paper we present the Persistent Grid Mapping (PGM), a novel framework for interactive terrain rendering that provides a screen-uniform tessellation of terrains in a view-dependent manner. The persistent grid, which covers the entire screen, is triangulated with respect to the rendering capabilities of the graphics hardware and cached in video memory. The GPU maps each vertex of the persistent grid onto the terrain. Such perspective mapping of the persistent grid remeshes the visible region of the terrain in a view-dependent manner with local adaptivity. Our algorithm maintains multiple levels of the elevation and color maps to achieve a faithful sampling of the viewed region. The rendered mesh ensures the absence of cracks and degenerated triangles that may cause the appearance of visual artifacts. In addition, an out-of-core support is provided to enable the rendering of large terrains that exceed the size of texture memory. PGM algorithm provides high quality images at steady frame rates.

1 Introduction

Interactive rendering of terrain datasets has been a challenging problem in computer graphics for decades. Terrain geometry is an important component of various visualization and graphics applications from computer games to professional flight simulators. The need for accurate visualization has led to the generation of large terrain datasets that exceed the interactive rendering capabilities of current graphics hardware. Moreover, the rendering of dense triangulations can lead to undesirable aliasing artifacts. Therefore, level-of-detail rendering seems to provide an appropriate framework for interactive rendering of large terrain datasets.

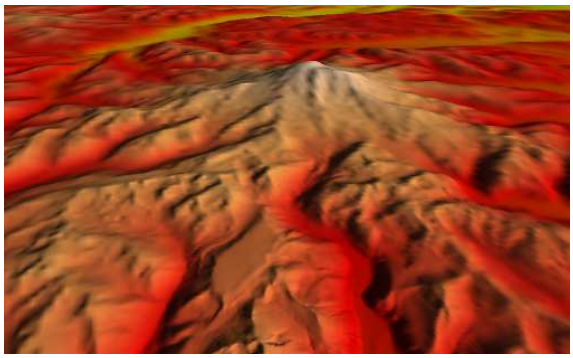


Figure 1. View of the Puget Sound area rendered by PGM algorithm.

Several level-of-detail based approaches have been developed for interactive terrain rendering. Traditional multiresolution and view-dependent rendering algorithms rely on the CPU to extract the appropriate levels of detail which are sent to the graphics hardware for rendering at each frame. In these approaches the CPU often fails to extract the frame's geometry from large datasets within the duration of one frame. In addition, the communication between the CPU and the graphics hardware often forms a severe transportation bottleneck. These limitations usually result in non-uniform or unacceptable low frame rates.

Current graphics hardware provides common functionality for both vertex and fragment processors. This functionality is useful for generating various effects such as displacement mapping, water simulation, and more [16]. Such significant improvement in graphics hardware necessitates the development of new algorithms and techniques to utilize the introduced processing power and programmability.

In this paper we present Persistent Grid Mapping (PGM),

a novel framework for interactive terrain rendering (Figure 1) which overcomes the drawbacks of previous approaches. At each frame, our algorithm samples the visible region of the terrain in a view-dependent manner to generate and render a new mesh that matches the rendering capabilities of the system's graphics hardware. The sampling is performed on the terrain base plane (at $z = 0$) by fast ray shooting, which is executed by the vertex processor and takes into account screen-space projection error. The vertex processor extracts the height component of the terrain from elevation maps at the place where the ray hits the terrain base plane. Then, it assigns the height value to the appropriate vertex of the sampled mesh. A similar procedure is executed in the fragment processor to extract the color components for each fragment. Our algorithm maintains multiple levels of the elevation and color maps to achieve faithful sampling of the visible region. In addition, an out-of-core support is provided to enable the rendering of large terrains that exceed the texture memory size.

PGM algorithm provides a number of advantages over previous terrain rendering schemes:

- **Efficiency:** Each vertex requires independent simple processing within the inherently parallel and powerful GPU. This leads to efficient rendering of terrain datasets.
- **System load balancing:** Most of the computations are done on the GPU, instead of the frequently overloaded CPU.
- **Steady frame rates:** The frame rates are steady and independent of terrain size as a result of rendering meshes of the same size at each frame. The complexity of the rendered mesh can adapt to match the capability of the graphics hardware.
- **Effortless levels of detail:** The perspective ray shooting remeshes the visible region of the terrain in a view-dependent manner. Such a scheme achieves effortless level-of-detail rendering with accurate screen-space projection error without any off-line preprocessing.
- **No culling required:** Since our algorithm sends the GPU only a mesh that resembles the view frustum, no view-frustum culling is required.
- **No visual artifacts:** Our rendered mesh ensures the absence of cracks (watertight) and sliver/degenerate triangles that may cause the appearance of visual artifacts.
- **Efficient out of core:** Our out-of-core scheme encodes various-level maps into one texture to avoid multiple texture binding by the CPU.

In the rest of this paper, we first overview the related work in the area of terrain rendering with emphasis on GPU-based algorithms. Then, we present our algorithm for terrain rendering and its out-of-core support followed by implementation details and results. Finally, we draw some conclusions and suggest directions for future work.

2 Previous work

In this section we overview closely related work in level-of-detail terrain rendering. We categorize these approaches based on the hardware orientation of their processing.

CPU-oriented processing: These algorithms purely rely on CPU computations and random-access memory to select the appropriate geometry which is sent to the graphics hardware at each frame. Some of these algorithms have been developed specifically for terrain rendering while others have targeted general 3D models.

Irregular meshes algorithms represent terrains as triangulated meshes. They usually utilize temporal coherence and manage to achieve the best approximation of the terrain for a given triangle budget. However, these algorithms require to maintain the mesh adjacency and to validate refinement dependences at each frame. Some hierarchies use Delaunay triangulation to constrain the mesh [8, 36], while others use less constrained multiresolution hierarchies with arbitrary connectivity [10, 14, 19, 31].

Regular level-of-detail hierarchies are usually more discrete as a result of working on regular grids or quadtrees. To simplify the memory layout and accelerate the mesh traversal they use a restricted quadtree triangulation [2, 34], triangle bintrees [13, 27], hierarchies of right triangles [15], or longest edge bisection [28]. However, updating the mesh at each frame prevents the use of geometry caching and efficient rendering schemes.

To reduce CPU load and utilize efficient rendering schemes several approaches partition the terrain into patches at different resolutions. Then, at real-time the appropriate patches are selected, stitched together, and sent for rendering [18, 34, 35]. Cignoni et al. [7] and Yoon et al. [42] have developed similar approaches for general 3D models. The main challenges are to select the appropriate patches quickly and to stitch their boundaries seamlessly. Although there are no geometric modifications, the communication between the CPU and the graphics hardware forms a serious bottleneck.

Utilizing video cache: To overcome the communication bottleneck several algorithms that utilize geometry cache have been introduced. Some algorithms [5, 6, 24, 26, 40] cache triangulated regions in video memory, while others use triangle strips to maximize the efficiency of the cache [20]. Terrains are often accompanied by huge tex-

ture (color) maps. Tanner et al. [39] have introduced the texture clipmaps hierarchy, while Döllner et al. [12] have introduced more general texture hierarchies to handle these texture maps. These caching techniques enable fast transition of geometry and texture to the graphics hardware. However, the cache memory is limited, thus large datasets may still involve a communication overhead between the CPU and the cache memory.

GPU-oriented processing: Frameworks for programmable graphics hardware have been suggested by [11, 17, 25, 29, 32]. Although these approaches are not implemented for the GPU, they are based on novel designs which prefer many simple computations over a few complicated ones.

The advances in graphics hardware programmability have led to the development of various algorithms. Losasso et al. [30] and Bolz and Schröder [4] use the fragment processor to perform mesh subdivision. Southern and Gain [38] use the vertex processor to interpolate different resolution meshes in a view-dependent manner. Wagner [41] and Hwa et al. [21] use GPU-based geomorphs to render terrain patches at different resolutions. Schneider and Westermann [37] reduce the communication time by using progressive transmission of geometry.

Several approaches are based on rendering a uniform grid. Kryachko [23] uses vertex texture displacement for realistic water rendering, while Johanson [22] projects a grid onto a water surface. However, both these algorithms focus on generating water effects rather than sampling datasets. Dachsbacher and Stamminger [9] modify the grid in procedural manner according to the camera parameters and the terrain features.

The geometry clipmaps algorithm [1] stores the surface triangulation layout in a view-dependent manner. At each frame, the visible part of the triangulation is sent to the GPU and is modified according to the uploaded height and color maps. However, this algorithm does not perform local adaptivity, and the transition between levels of detail is not smooth and may result in cracks. These cracks are resolved by inserting degenerate triangles which may lead to visual artifacts.

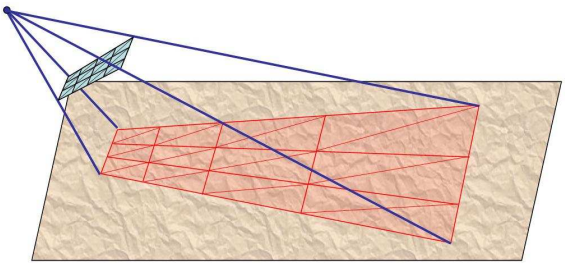


Figure 2. Mapping of the persistent grid onto the terrain base plane.

3 Our algorithm

In this section we present our algorithm for interactive terrain rendering that leverages advanced features of current graphics hardware, such as programmable vertex and fragment processors, displacement mapping, and geometry caching. In an initial step PGM algorithm covers the viewport with a triangulated grid which is cached on the GPU. The programmable GPU maps each vertex of the grid onto the terrain base plane (see Figure 2). Then, it fetches the height value from the texture memory and assigns it to the mapped vertex.

The throughput of current GPUs is enough to cover a framebuffer with pixel-size triangles at interactive rates. In addition, vertex processing is catching up with the pixel fill rate. Therefore, a screen-uniform tessellation of the terrain, where all triangles are nearly pixel-size, would efficiently utilize these advances. To achieve such tessellation, we generate a uniform triangulated grid that covers the entire screen or working window. The resolution of the generated grid is determined by the rendering capabilities of the available graphics hardware. Since we seldom resize the working window, the generated grid remains constant over a large number of frames (and usually over the entire navigation session). For that reason, we shall refer to this triangulated grid as the *persistent grid*.

The CPU generates the persistent grid and aligns it with the viewport. The distribution of the grid vertices is not constrained – it could be uniform, non uniform, or even a general irregular grid. For example, a focus-guided triangulation of the visualized terrain could be achieved by using a spherical persistent grid with parametrized subdivision.

The persistent behavior enables the caching of the grid in video memory which is available in even modest graphics hardware. Such caching manages to remove the severe communication bottleneck between the CPU and the GPU by transmitting the generated persistent grid only once and not at each frame. In addition, it eliminates annoying variations on frame rates by rendering the same number of polygons – the persistent grid – at each frame.

3.1 Persistent grid mapping

The mapping of the persistent grid onto the terrain base plane is performed by shooting rays from the viewpoint through the vertices of the persistent grid and computing their intersection with the terrain base plane (at $z = 0$). This mapping is similar to the projected grid concept introduced in [22]. The vertex G_{ij} on the persistent grid is mapped to the point P_{ij} on the terrain base plane by using Equation 1, where V is the viewpoint, and v_z and g_z are the z coordinates of the viewpoint and the grid vertex G_{ij}, respectively.

$$P_{ij} = V + \frac{v_z}{v_z - g_z}(G_{ij} - V) \quad (1)$$

Programmable vertex processor in current graphics hardware provides the ability to access the graphics pipeline and alter vertex properties. The simplicity and compactness of Equation 1 facilitate the implementation of ray shooting within the vertex processor. Figure 3 illustrates the translation of the grid vertex G_{ij} to the terrain vertex T_{ij}, which is performed by executing the following steps:

1. The coordinates of the grid vertex G_{ij} and the viewpoint V are used to compute P_{ij} on the terrain base plane (Equation 1).
2. The x and y coordinates of P_{ij} are used to fetch the height value from the elevation map.
3. The updated terrain vertex T_{ij} is composed of the x and y coordinates of P_{ij} and the fetched height value.

Then, the updated vertex continues its rendering journey. In the fragment processor similar computations are executed for each fragment to extract its color from the color map.

3.2 Camera restriction

The mapping of the persistent grid onto the terrain base plane works fine whenever the camera aims toward the plane. However, when this constraint is violated, rays that are parallel or upward do not intersect the terrain base plane. One could omit these rays, thus discard all polygons mapped by these rays. This naive solution has two problems. The first is that the disregarded rays waste the GPU's computational power without contributing to the final image. The second and severer problem is that the sampled region is offset from the visible region. An extreme case of this offset problem may lead to the missing of close peaks when the viewer is flying low as shown in Figure 4(a).

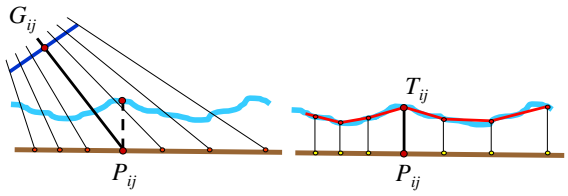


Figure 3. The translation of grid vertices to terrain vertices. (a) Mapping the grid vertices onto the terrain base plane. (b) Fetching height values from the elevation map.

To resolve this problem, we use an additional camera – the *auxiliary camera* – which is designated for sampling terrain data and generating mesh geometry. The rendering of the generated geometry is performed with respect to the parameters of the *main camera*. The view direction of the auxiliary camera is computed by Equation 2, where A_{aux} and A_{main} are the angles between the horizontal plane and the view direction of the auxiliary and main camera, respectively, and FOV is the field of view of the main camera.

$$A_{aux} = max(A_{main},\ FOV/2 + \epsilon),\ \epsilon \to 0 \quad (2)$$

As can be seen from Equation 2, the auxiliary camera has the same position and orientation as the main camera as long as no parallel or upward rays are created. However, when the main-camera parameters change in a way that such rays are created, we restrict the view direction of the auxiliary camera. This restriction ensures that all rays intersect the terrain base plane, see Figure 4(b).

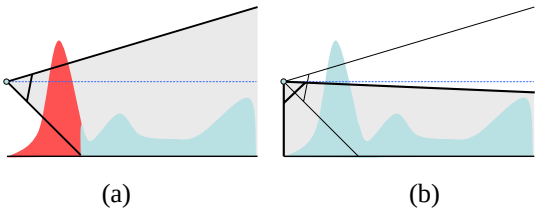
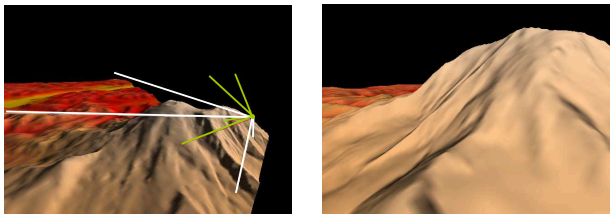


Figure 4. Camera restriction. (a) A peak missed by the main camera (in red). (b) The use of an auxiliary camera.

This solution inherently solves the first problem. The offset problem is also handled appropriately since the auxiliary camera samples data of the area starting just below the viewpoint and ending at the far horizon (upper rays are almost parallel to the terrain base plane by using $\epsilon \to 0$). This observation is true when the FOV angle of the main camera is at least 90 degrees, otherwise we can extend the FOV of the auxiliary camera to 90 degrees. Note that the auxiliary camera retains the screen-space projection error of the main camera for the region shared by the two cameras. Figure 5 shows a close peak rendered using our solution. The view-frustums of the main and auxiliary cameras are colored in green and white, respectively.

3.3 Level-of-detail rendering

Previous approaches for level-of-detail terrain rendering usually rely on off-line-constructed multiresolution hierarchies of geometry. Then, at run-time these hierarchies are used to extract the appropriate level of detail in a view-dependent manner. In contrast, PGM does not require any



(a) (b)

Figure 5. A close peak. (a) Side view of the main (green) and auxiliary (white) cameras. (b) The resulting image.

preprocessing construction of such multiresolution hierarchies to support level-of-detail rendering. The perspective mapping of the persistent grid onto the terrain generates different levels of detail in an automatic manner. Similar triangles on the persistent grid are mapped to different-size triangles on the terrain – close-to-viewer triangles occupy smaller regions than far-from-viewer triangles. The persistent grid mapping algorithm provides smooth transition between adjacent levels of detail as shown in Figure 6.

View-dependent rendering algorithms try to represent objects with respect to their visual importance in the final image. Approximated screen-space projection error has been used to guide the selection of various resolutions over different regions in the object space [3, 19, 31]. In contrast, PGM relies on the persistent grid, which inherently provides quality screen-space projection error to determine the appropriate level of detail for each region. When the vertices are lying on the terrain base plane, the screen-space projection error is absolutely accurate. However, due to the vertices displacement with respect to the terrain height, screen-space projection error slightly loses its accuracy (see Figure 7).

Reducing the number of polygons sent to the graphics hardware by culling invisible geometry has been used to

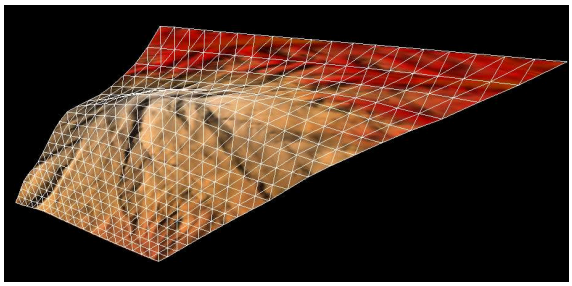


Figure 6. Smooth transition between adjacent levels of detail (side-view of a mapped persistent grid of size 20x20).

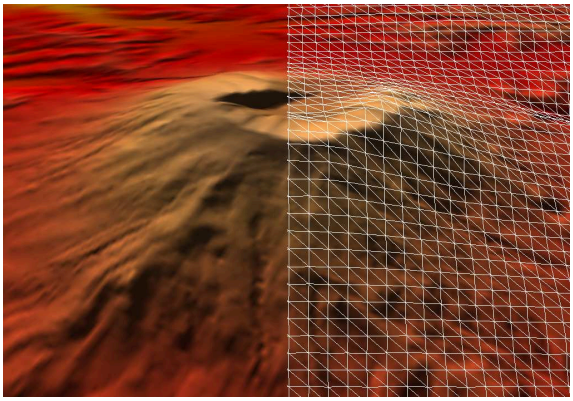


Figure 7. The screen-space projection error resulting from a uniform triangulated grid (of size 50x50).

accelerate the rendering of large models. View-frustum culling, back-face culling, and sophisticated occlusion culling are used to achieve this goal. In contrast, our approach does not explicitly apply any culling since the mapped persistent grid delimits and usually resembles the view frustum.

3.4 Terrain sampling

The mapping procedure displaces the persistent grid according to the visible region of the terrain. However, this procedure often fails to generate a faithful image of the terrain as a result of using sparse sampling with narrow filters. Figure 8 shows an example of missing vital details of the rendered terrain. Practically, one could resolve this problem by using more samples with wider sampling filters. However, fetching data from the texture memory is not fast enough in current hardware. Furthermore, using wide filters requires all the visible part of the maps to be uploaded in texture memory which is impossible for large terrains.

To overcome such a limitation, we use multiple-level texture pyramids at successive powers of two (similar to mipmaps) to cache elevation and color maps. These texture pyramids are used to achieve better sampling of the mapped

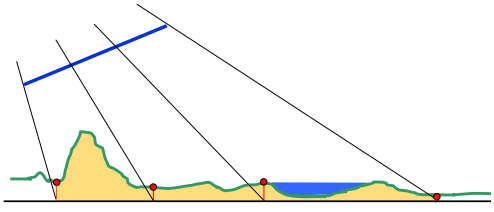


Figure 8. Missing hill and lake.

persistent grid. Since these multiple-level pyramids are similar to mipmaps, we could let the hardware construct them. Then, at the vertex processor our algorithm could determine 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. In Section 4 we introduce our out-of-core scheme that handles large terrains.

The multiple-level texture pyramids are constructed once in real-time by the CPU before being transmitted to the texture memory. Starting with the original (most detailed) level, each 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 averaging the four corresponding pixels of the previous level.

The mapping procedure selects the height and color values from the different levels of the hierarchies based on the geometric level of detail at the mapped vertex. Therefore, we access the texture level, which pixel size corresponds to the distance r between the vertex v and its farthest adjacent neighbor (see Figure 9). It is easy to determine the adjacent neighbors of a vertex since the mapping of the persistent grid preserves connectivity. For efficiency, we can approximate r based on the height h of the viewpoint, the distance d between the viewpoint and the vertex v, and q which is the distance d projected onto the base plane. Equation 3 presents an approximation of r, where α is the angle between adjacent rays and β is the angle between the current ray and the normal of the base plane. Since the variation of α is very small for different rays, we can assume that α is constant over the entire frame.

$$r = \frac{dsin\alpha}{cos(\beta+\alpha)} = \frac{d^2 sin\alpha}{hcos\alpha - qsin\alpha} \quad (3)$$

The persistent grid mapping cannot maintain temporal continuity for both geometry and color for far-from-viewer regions as a result of sampling from coarse levels of elevation and color maps. This leads to popping effects that appear when samples of the same region slightly move over consecutive frames and fall over adjacent pixels. Such popping effects are even more evident when the adjacent pixels

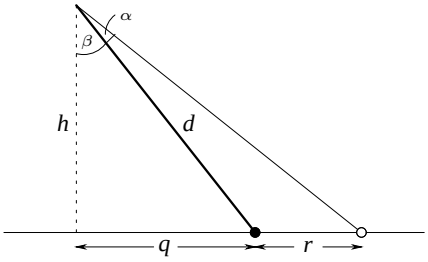


Figure 9. The distance r determines the level of detail at vertex v.

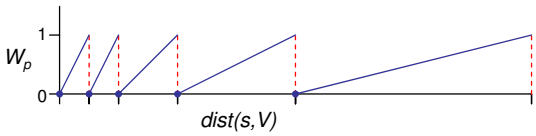


Figure 10. The graph of the parent's weight.

belong to different texture levels. To prevent such a drawback, we linearly interpolate the four adjacent pixels and the parent pixel (from the coarser texture level) of the sampling point s. Since these adjacent pixels may have different parents, we first interpolate four adjacent pixels from the coarser level and refer to the resulted value as a parent pixel. The weight of each pixel depends on its distance from s and the contribution of the parent pixel depends on how close we are to switch to the next level. The weight of the parent W_p is calculated by using Equation 4, where $dist(s,V)$ is the distance of the sampling point s from the projected viewpoint V and lod is the level of detail at s.

$$W_p = \frac{dist(s,V)}{2^{lod-1}} - 1 \quad (4)$$

Figure 10 presents the graph of Equation 4 for the five finest levels of detail. Figure 11 shows a terrain rendered with and without height and color interpolations.

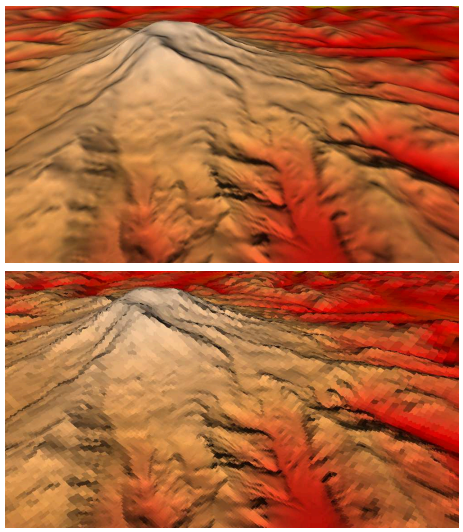


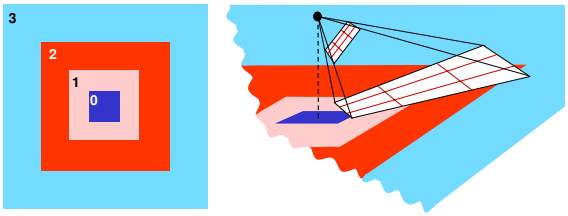
Figure 11. A terrain rendered with (up) and without (down) interpolations.

4 Out-of-core rendering

So far our algorithm has assumed that the elevation and color maps fit in texture memory. However, current terrain datasets often exceed the capacity of typical texture memory. The handling of large datasets includes two uploading stages – from external media into main memory and from main memory into texture memory. Uploading data from external media into main memory is widely studied [21, 28] and it is out of the scope of this paper. Limited size of texture memory on current hardware calls for designing a scheme for uploading data from main memory to texture memory.

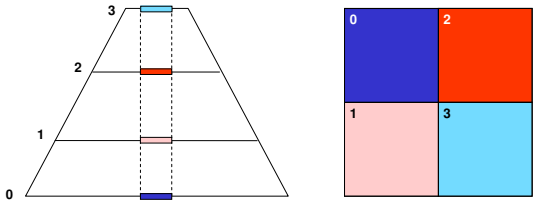
Our texture memory manager starts by constructing the multiple-level pyramids on the fly and maintains in texture memory the portions of data necessary for rendering the next frame. The upload from main memory into texture memory is performed by fetching nested clipmaps centered about the viewpoint [1, 39] as shown in Figure 12. Even though these clipmap levels occupy the same memory size, they cover increasing regions of the terrain. Each extracted level is an independent texture that requires separate binding at the CPU. However, the CPU cannot predict the level of detail of the mapped vertices since the selection of level of detail is determined at the vertex processor.

One could resolve this limitation by binding several textures and selecting the appropriate texture map in real-time at the vertex processor. However, the number of textures that can be bound at the same time is limited and vary for different graphics cards. Moreover, the selection of the appropriate texture map involves branching operations which are expensive in current GPUs. Therefore, we apply a technique that uses multiple texture maps without rebinding or branching. We achieve this by laying all the different texture levels into one large texture similar to the idea of Texture Atlases [33]. They tile up uniformly because of their equal sizes as shown in Figure 13. In such a scheme, the level of detail of a vertex determines the appropriate tile and the texture coordinates are calculated accordingly.



(a) (b)

Figure 12. Clipmaps. (a) Nested clipmaps centered about the viewpoint. (b) The persistent grid is mapped across several clipmaps.



(a) (b)

Figure 13. The layout of the different texture levels into one texture map. (a) The texture pyramid holds the entire terrain in decreasing resolutions. (b) All clipmaps layed on a texture atlas.

In traditional texture atlases, triangle's texture coordinates do not span across different sub-tiles of the atlas. However, in our algorithm adjacent vertices may fall in two consecutive levels of detail, thus the generated fragments fall across different tiles which results in incorrect pixels (see Figure 14). Such incorrectness occurs due to the automatic interpolation done in the fragment processor. To resolve this problem, the fragment processor explicitly calculates the correct texture coordinates for each fragment based on parameters provided by the vertex processor.

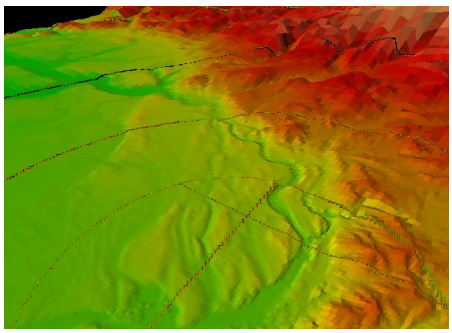


Figure 14. Incorrect pixels resulted by automatic interpolation in the fragment processor.

The changes in view parameters, such as viewpoint, look-at, or up-vector, 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]. Since the resolution of clipmap levels exponentially decreases, the area that needs to be updated gets smaller. Moreover, coarse levels are seldom updated. The updated areas of the tiles form an F-shape since all the tiles lay in one texture (see Figure 15).

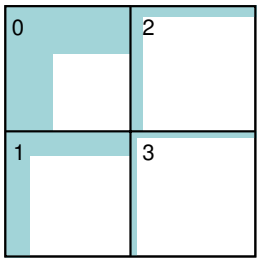


Figure 15. The F-shape modification.

Persistent Grid Size	Triangles Number (K)	Frames/second		
		Multiple Textures	2 Frag. Fetches	1 Frag. Fetch
150x600	180	83	177	226
200x800	320	56	119	133
430x430	370	54	112	120
300x900	540	37	78	81
600x600	720	25	59	59
400x1600	1280	16	35	35

Table 1. Comparison of various settings.

5 Implementation and results

We have implemented our algorithm in C++ and Cg for Microsoft Windows operating system. Then, we have tested our implementation on various datasets and have received encouraging results.

If we assume that the terrain dataset entirely fits in texture memory, then the implementation of our algorithm is simple. The persistent grid is generated and stored in video memory, and the mapping procedure is performed by several Cg instructions within the GPU. In such a case, the hardware-generated mipmaps can be used as the texture pyramids. The need for out-of-core support requires the construction of the clipmap pyramids which is performed by the CPU. Then these pyramids are transfered to texture memory. The updating of the clipmaps is performed in an active manner by transferring several rows/columns at each frame.

Fetching data from texture memory is an expensive operation, but four floats can be fetched by a single operation. Careful layout of the data in texture memory enables significant reduction in the number of fetches. We perform a single vertex fetch to obtain the data required to determine the height of a vertex (including interpolations). For each fragment we perform two fetches to interpolate 32-bit colors. By performing only a single fetch we can reduce the color quality (16-bit colors) or alternatively use partial interpolation (ignoring parent contribution). Using a single fetch per fragment compromises image quality for frame rates and vice versa.

The results we report in this section have been achieved on a PC machine with an AMD Athlon 3500+ processor, 1GB of memory, and an NVIDIA GeForce 7800 GTX graphics card with 256MB texture memory. We use a 16Kx16K grid of the Puget Sound area as our main dataset.

Table 1 summarizes the performance of our PGM algorithm. The first two columns show the resolution and triangle count of different-size persistent grids. Note that we report results for square as well as rectangular persistent grids. The remaining columns show the frame rates under various settings. The *Multiple Textures* column shows the frame rates achieved by using multiple binding (with branching) for the different texture levels. The fourth and fifth columns report the frame rates when performing two and one fetches per fragment, respectively.

As can be seen, PGM manages to achieve high frame rates while generating quality images. Our algorithm provides better adaptivity than previous approaches since it can select the resolution of the persistent grid that matches the rendering capabilities of the graphics hardware. The most related approach to this work is geometry clipmaps since both utilize the GPU to accelerate the rendering of large terrains. Asirvatham and Hoppe [1] report a frame rate of 87 frames/sec when rendering approximately 1.1M Δ/frame on NVIDIA GeForce 6800 GT. After view-frustum culling the number of rendered triangles reduces to about 370K Δ/frame which is equivalent to our persistent grid of size 430x430. Using current comparable hardware (GeForce 7800 GTX) they would approximately obtain 104 frames/sec which is similar to the result of PGM using a persistent grid of size 430x430 with two fragment fetches and interpolations. However, as a result of using a smoother mesh we achieve local adaptivity without cracks or degenerate triangles.

Component	Vertex Processor		Fragment Processor	
	msec	%	msec	%
Computations	2.56	37	0.16	6
Fetches	2.30	33	0.66	21
Interpolations	2.05	30	3.16	73
Overall	6.91	100	3.98	100

Table 2. Processing times for the various steps of the rendering process.

Table 2 presents the processing times for the various steps of the rendering process for both the vertex and fragment processors. The times refer to a single frame when rendering a grid of size 450x450 (with two fetches in the fragment processor). By analyzing the overall processing

time, we can conclude that the performances of our algorithm are bounded by the vertex processor.

Table 3 shows the performance of our out-of-core scheme using various tile sizes and three navigation patterns – fast, medium and slow motion. The tile (clipmaps) sizes in pixels appear in the first column. The *Entire Texture* column reports the time, in *msec*, of updating the entire texture at each frame. Such a scenario results in unacceptable frame rates. The *F-shape Update* columns report the time for the different navigation patterns. The fast-motion navigation implies extensive updates of the tiles, whereas the slow-motion navigation requires only minor updates at each frame.

Tile Size	Entire Texture (msec)	F-shape Update (msec)		
		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

Table 3. Out-of-core performances.

Figures 16 and 17 show two snapshots of different regions of the Puget Sound dataset using a persistent grid of size 200x800. As can be seen even for modest grid sizes we achieve quality images.

6 Conclusions and future work

We have presented a novel framework for rendering large terrain datasets at interactive rates by utilizing advanced features of current graphics hardware. The view-dependent level-of-detail rendering is performed in an automatic fashion by the perspective mapping of the persistent grid onto the terrain. The combination of nearly accurate screen-space projection error and level-of-detail rendering results in quality final images. Moreover, the use of a mesh as the final rendering representation of the viewed surface eliminates cracks and artifacts and allows the use of typical illumination models. Since the complexity of the rendering mesh is constant (as result of using the same persistent grid), the rendering frame rates are smooth and steady.

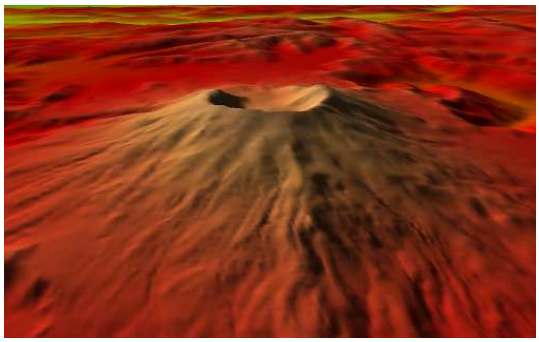


Figure 16. A snapshot from the Puget Sound dataset.

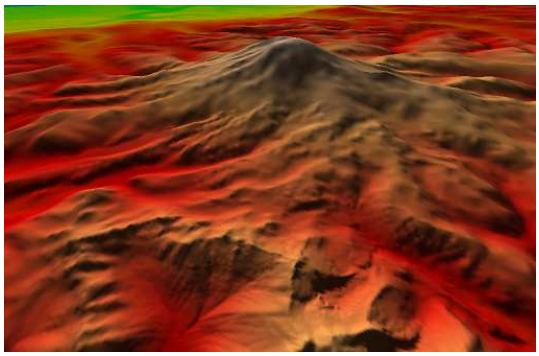


Figure 17. A snapshot performed by PGM.

Our out-of-core scheme allows rendering of large terrain datasets. It avoids rebinding of textures by embedding all the levels of detail (for elevation and color maps) in the same texture. The textures are efficiently updated by using F-shape modifications.

We expect that the current trend in improving the vertex processor will continue and directly contribute to the performance of PGM algorithm. We see the scope of future work in improving the metrics used in the construction of the multiple-level texture pyramids. It is important to study the feasibility of using PGM for image-based and volume rendering.

Acknowledgments

Our research and particularly this work is supported by the Lynn and William Frankel Center for Computer Sciences, the Israel Ministry of Science and Technology, and the Tuman fund. In addition, we would like to thank Yoel Shoshan for his assistance with the efficient implementation.

References

[1] A. Asirvatham and H. Hoppe. Terrain rendering using GPU-based geometry clipmaps. *GPU Gems 2*, pages 27–45, 2005.

[2] X. Bao, R. Pajarola, and M. Shafae. SMART: An efficient technique for massive terrain visualization from out-of-core. In *Proceedings of Vision, Modeling and Visualization '04*, pages 413–420, 2004.

[3] J. Blow. Terrain rendering at high levels of detail. In *Game Developers Conference*, 2000.

[4] J. Bolz and P. Schröder. Evaluation of subdivision surfaces on programmable graphics hardware. *Submitted*, 2005.

[5] 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*, 22(3):505–514, 2003.

[6] P. Cignoni, F. Ganovelli, E. Gobbetti, F. Marton, F. Ponchio, and R. Scopigno. Planet-sized batched dynamic adaptive meshes (P-BDAM). In *Proceedings of Visualization '03*, pages 147–155, 2003.

[7] P. Cignoni, F. Ganovelli, E. Gobbetti, F. Marton, F. Ponchio, and R. Scopigno. Adaptive tetrapuzzles: efficient out-of-core construction and visualization of gigantic multiresolution polygonal models. *ACM Trans. Graph.*, 23(3):796–803, 2004.

[8] D. Cohen-Or and Y. Levanoni. Temporal continuity of levels of detail in delaunay triangulated terrain. In *Proceedings of Visualization '96*, pages 37–42, 1996.

[9] C. Dachsbacher and M. Stamminger. Rendering procedural terrain by geometry image warping. In *Eurographics Symposium in Geometry Processing*, pages 138–145, 2004.

[10] L. De Floriani, P. Magillo, and E. Puppo. Building and traversing a surface at variable resolution. In *Proceedings of Visualization 97*, pages 103–110, 1997.

[11] M. Doggett and J. Hirche. Adaptive view dependent tessellation of displacement maps. In *HWWS '00: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 59–66, 2000.

[12] J. Döllner, K. Baumann, and K. Hinrichs. Texturing techniques for terrain visualization. In *Proceedings of Visualization '00*, pages 227–234, 2000.

[13] M. Duchainear, M. Wolinsky, D. Sigeti, M. Miller, C. Aldrich, M., and Mineev-Weinstein. ROAMing terrain: Real-time optimally adapting meshes. In *Proceedings of Visualization '97*, pages 81–88, 1997.

[14] J. El-Sana and A. Varshney. Generalized view-dependent simplification. *Computer Graphics Forum*, 18(3):83–94, 1999.

[15] W. S. Evans, D. G. Kirkpatrick, and G. Townsend. Right-triangulated irregular networks. *Algorithmica*, 30(2):264–286, 2001.

[16] G. Gerasimov, F. Fernando, and S. Green. Shader model 3.0 using vertex textures. *White Paper*, 2004.

[17] S. Gumhold and T. Hüttner. Multiresolution rendering with displacement mapping. In *HWWS '99: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 55–66, 1999.

[18] L. Hitchner and M. McGreevy. Methods for user-based reduction of model complexity for virtual planetary exploration. In *SPIE 1913*, pages 622–636, 1993.

[19] H. Hoppe. Smooth view-dependent level-of-detail control and its application to terrain rendering. In *Proceedings of Visualization '98*, pages 35–42, 1998.

[20] H. Hoppe. Optimization of mesh locality for transparent vertex caching. In *Proceedings of SIGGRAPH '99*, pages 269–276, 1999.

[21] L. M. Hwa, M. A. Duchaineau, and K. I. Joy. Adaptive 4-8 texture hierarchies. In *Proceedings of Visualization 2004*, pages 219–226, 2004.

[22] C. Johanson. Real-rime water rendering - introducing the projected grid concept. *Master's thesis, Lund Univ.*, 2004.

[23] Y. Kryachko. Using vertex texture displacement for realistic water rendering. *GPU Gems 2*, pages 283–294, 2005.

[24] R. Lario, R. Pajarola, and F. Tirado. HyperBlock-QuadTIN: Hyper-block quadtree based triangulated irregular networks. In *Proceedings of IASTED VIIP*, pages 733–738, 2003.

[25] B. S. Larsen and N. J. Christensen. Real-time terrain rendering using smooth hardware optimized level of detail. In *WSCG*, 2003.

[26] J. Levenberg. Fast view-dependent level-of-detail rendering using cached geometry. In *Proceedings of Visualization '02*, pages 259–266, 2002.

[27] P. Lindstrom, D. Koller, W. Ribarsky, L. F. Hodges, N. Faust, and G. A. Turner. Real-time, continuous level of detail rendering of height fields. In *Proceedings of SIGGRAPH '96*, pages 109–118, 1996.

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

[29] F. Losasso and H. Hoppe. Geometry clipmaps: terrain rendering using nested regular grids. *ACM Trans. Graph.*, 23(3):769–776, 2004.

[30] F. Losasso, H. Hoppe, S. Schaefer, and J. Warren. Smooth geometry images. In *Eurographics Symposium in Geometry Processing*, pages 138–145, 2003.

[31] D. Luebke and C. Erikson. View-dependent simplification of arbitrary polygonal environments. In *Proceedings of SIGGRAPH '97*, pages 198 – 208, 1997.

[32] K. Moule and M. D. McCool. Efficient bounded adaptive tessellation of displacement maps. In *Proceedings of Graphics Interface*, pages 171–180, 2002.

[33] NVIDIA. Improve batching using texture atlases. *SDK White Paper*, 2004.

[34] R. Pajarola. Large scale terrain visualization using the restricted quadtree triangulation. In *Proceedings of Visualization '98*, pages 19–26, 1998.

[35] A. Pomeranz. Roam using triangle clusters (RUSTiC). *Master's thesis*, 2000.

[36] B. Rabinovich and C. Gotsman. Visualization of large terrains in resource-limited computing environments. In *Proceedings of Visualization '97*, pages 95–102, 1997.

[37] J. Schneider and R. Westermann. GPU-friendly high-quality terrain rendering. *Journal of WSCG*, 14(1-3):49–56, 2006.

[38] R. Southern and J. Gain. Creation and control of real-time continuous level of detail on programmable graphics hardware. *Computer Graphics Forum*, 22(1):35–48, 2003.

[39] C. C. Tanner, C. J. Migdal, and M. T. Jones. The clipmap: a virtual mipmap. In *Proceedings of SIGGRAPH '98*, pages 151–158, 1998.

[40] T. Ulrich. Rendering massive terrains using chunked level of detail control. In *Proceedings of SIGGRAPH' 2002*, 2002.

[41] D. Wagner. Terrain geomorphing in the vertex shader. *ShaderX2: Shader Programming Tips & Tricks with DirectX 9*, 2004.

[42] S.-E. Yoon, B. Salomon, and R. Gayle. Quick-VDR: Out-of-core view-dependent rendering of gigantic models. *IEEE Transactions on Visualization and Computer Graphics*, 11(4):369–382, 2005.