Terrain Rendering with the Combination of Mesh Simplification and Displacement Mapping

Zsolt Fehér*
Supervised by: Zoltán Prohászka†

Department of Control Engineering and Information Technology
Budapest University of Technology and Economics
Budapest / Hungary

Abstract

Today's graphics hardware is not capable of displaying arbitrarily detailed terrains in real–time. Above a certain number of triangles rendering becomes too slow, frame rate drops below 30. Known methods – such as Level of Detail (LOD) or Realtime Optimally-Adapting Meshes (ROAM) – simplify the triangle meshes to maintain speed. With fewer triangles real–time speeds are possible, but it decreases the visual quality. This paper presents a fast, real-time method that combines triangle mesh simplification with pixel shader displacement mapping. This method first builds up an approximate low resolution triangle mesh and applies displacement mapping on that. In the pixel shader it computes the intersection of rays and the terrain. For the best result, it combines linear search with secant search. The result is independent from the resolution of the heightmap and is capable of displaying terrains without decreasing detail.

Keywords: Terrain rendering, Real-time, Displacement mapping, Ray tracing, Linear search, Secant method

1 Introduction

Usually height values of terrains are stored in heightmaps, i.e. two dimensional grayscale textures. Above a certain resolution of the heightmap, too many polygons need to be displayed if we draw triangles between each height point. The result will not be real-time. For example, a heightmap with 2049 x 2049 points would mean more than 8 million triangles. At present, an average GPU could display it only below 30 fps. There are known methods, which simplify the triangle mesh. Most popular are Level of Detail (LOD) and Realtime Optimally-Adapting Meshes (ROAM). As they display fewer triangles, frame rate is higher and acceptable, but visual quality is reduced. It is also possible to display the land with displacement mapping. This could be faster and independent of the resolution of the heightmap, but also could be inaccurate.

*zsolt.feher.88@gmail.com
†prohaszka@iit.bme.hu

This paper presents a method that combines triangle mesh simplification with displacement mapping. It combines the advantages of both methods. With reduced number of triangles, high frame rate is possible. With displacement mapping there is no loss in detail. The triangle mesh helps displacement mapping to be faster and much more accurate. The algorithm first builds up an approximate low resolution triangle mesh above the real relief, then it uses displacement mapping on that. The pixel shader of the GPU determines the ray, and on a segment of the ray, it searches for the intersection with the real terrain. Minimizing inaccuracy, first it uses linear search, and finally refines the result with secant search. The result is fast, independent of the resolution of the heightmap, and there is no degradation in detail.

2 Related Work

Graphics hardware is only able to display a limited number of triangles in real-time. For acceptable frame rate and visual quality several algorithms have been developed in the last decades. These algorithms reduce the number of triangles without significantly decreasing the visual quality. An often used method is LOD [4] [15]. Simplest version divides the surface to quads. In the quad where the camera is, the resolution of terrain is not changed. In distant quads the resolution is highly reduced. The border between two quads is noticeable and could be annoying. Due to the different resolutions, gaps could appear.

Another solution is the ROAM [5]. This algorithm splits triangles into two smaller triangles if the difference between the original and the new triangles is too big, and merges neighboring triangles if the difference is small. This algorithm considers the variety of surfaces. Flat part of the terrain would only consist of few triangles. It also considers what is visible from the camera. Far parts are displayed with fewer triangles than near parts. Both algorithms have an annoying problem. When the resolution of the triangle mesh changes somewhere, this change is instant. The surface pops between two frames, which is easy to notice.

I developed a method that simplifies the triangle mesh.

First the terrain consists of a few big triangles. The algorithm recursively examines every triangle if it is necessary to split it into two equal triangles. A triangle won't be split if the difference between the original and the new one is small. The algorithm handles the variety of surfaces and the triangle mesh doesn't change during running. However gaps could appear on the surface, due to the different size of triangles. Figure 1 shows a simplified terrain with gaps produced by this algorithm. There exist methods to fill these gaps, but they require additional computation and still create difference from the original terrain.

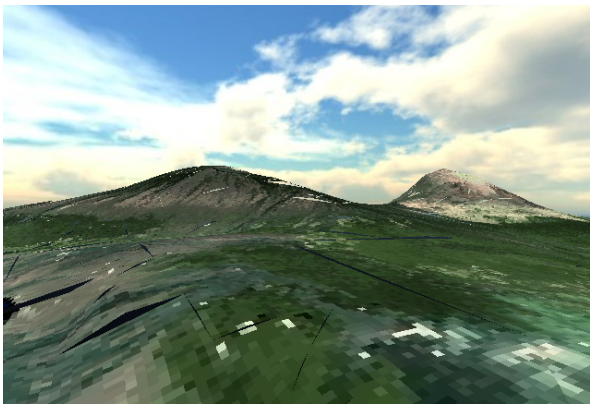

Figure 1: Simplified terrain with gaps

It is also possible to display a terrain without using triangle mesh. Using displacement mapping on a quad is capable of visualizing 3D surfaces. There are several methods that calculate the surface in the pixel shader [1], including parallax mapping, linear – binary search, secant method, cone stepping, sphere tracing etc. Displacement mapping could be really fast, due to the minimal polygon number. Unfortunately it could be also inaccurate.

The widely used per-vertex displacement mapping method (see [1]) differs from our approach, since it uses the vertex shader only, while our method uses both vertex and pixel shaders for displacement mapping.

There also exist a few approaches that combine triangle mesh with displacement mapping [2] [3]. In [2] a similar approach is used for vegetation visualization on orthographic landscape. Our approach is addressed for displacement mapping in general, for example to be used by visualizing reliefs. By our approach, the input is a homogeneous, high resolution height map, while [2] uses different datasets as the input of the visualizing algorithm. In [2], the determination of the height offset is not detailed, in our method it is calculated correctly and based on the height map.

3 Approximate Low Resolution Triangle Mesh

With basic displacement mapping we encountered a serious problem. Displacement mapping is usually used to increase the detail of a surface without using more polygons. Displacement mapping algorithms usually assume that the camera doesn't go below the original surface, and doesn't fly between bumps. In our application we would like to fly among the hills. If we draw the displaced terrain below the polygons, the terrain disappears when camera move below them. A better solution is to create the terrain above the polygons. Then the camera can fly between hills, but for horizontal and above horizon rays the terrain doesn't appear. When a ray doesn't intersect the original polygons, the pixel shader algorithm doesn't start calculating the intersection with terrain (see Figures 2and 3).

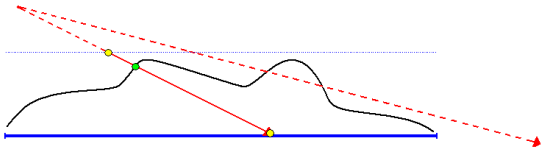

Figure 2: Ray doesn't intersect polygons

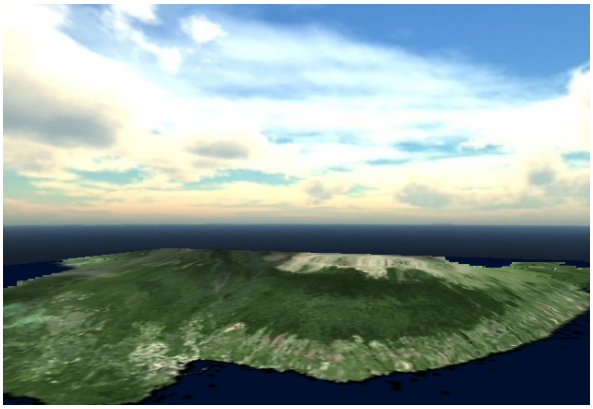

Figure 3: Terrain is not displayed above the base triangles (without using bounding box)

A known solution to this problem is the application of a bounding box. It builds up a box around the displaced surface, therefore every ray would intersect a polygon, and the pixel shader could determine the intersection with the terrain.

In the project we developed a faster method for this problem. It builds up an approximate low resolution triangle mesh above the real terrain in three steps. On this mesh it applies displacement mapping. Since the real terrain is very close to the mesh, the searches in pixel shader will be much faster.

3.1 First Step: Low Resolution Triangle Mesh

First the algorithm creates a low resolution triangle mesh. It uses height values from the height map. It doesn't use all pixels from height map, but skips a certain number of pixels. The resolution of the new mesh can be 16×16, 32×32, 64×64 or 128×128 quads. The optimal choice depends on the graphic card. On an Nvidia Geforce 8400 M the best result was observed at 32×32, but on an 8800 GT we got better outcome with 128×128. For arbitrary resolution, it is important that the resolution of the heightmap is $2^n + 1$.

3.2 Second Step: Push Up the Quads

For each quad[1], the algorithm examines the height values covered by a quad. It computes if a point is above the quad, then stores the highest among them. After the highest point above the quad is determined, the whole quad is being pushed upward to the maximum point. To do this, it is necessary to calculate the distance between the proper triangle's plane and the point. A simple method is to compute intersection between the plane and a vertical line. Place vector $\mathbf{r_0}$ of the line is at 0 height.

$$t = \frac{(\mathbf{p} - \mathbf{r_0}) \cdot \mathbf{n}}{\mathbf{v} \cdot \mathbf{n}} \tag{1}$$

where $\mathbf{p}$ is a point of the plane[2], $\mathbf{r_0}$ is a point of the line, $\mathbf{n}$ is the normal vector of the plane and $\mathbf{v}$ is the direction vector of the line. It is easy to compute the plane's normal vector from the vertices of the triangle.

$$\mathbf{n} = (\mathbf{b} - \mathbf{a}) \times (\mathbf{c} - \mathbf{a}) \tag{2}$$

where $\mathbf{a}$ and $\mathbf{b}$ are vertices of the triangle. Since the point of the line is at 0 height, parameter t is the exact height of the intersection. The difference between t and the real height value shows how far the quad should be raised. The algorithm stores in an array how each quad has been pushed up. This is used in step three.

3.3 Third Step: Fill Gaps

Each quad rose differently, scales of pushes are different at neighbors, therefore gaps appear between quads. These gaps should be removed. For every quad at every vertex, the algorithm examines the other (maximum three) vertices at that point and determines which one is the highest. The vertex (not the entire quad) should be pushed to the height of the highest vertex at that point. After every vertex raised to the proper height, there will be no gaps.

With these three steps, we got a low resolution terrain. The original terrain is never above the new surface but is very close to it.

[1] A quad consists of two triangles.

[2] In this case any vertex of the triangle

4 Displacement Mapping on Approximate Triangle Mesh

If we apply displacement mapping on the approximate triangle mesh, we get better results than using it with bounding box, because the triangle mesh is very near to the real surface. The relevant part of the ray is shorter, and with linear search the intersection could be found mostly within a few steps.

The heightmap texture stores height values between 0 and 255. These will be normalized to the range 0 and 1 on the GPU. In the rest of the article we assume that both $[U\ W\ H]$ coordinates are normalized to 0 and 1.

During basic displacement mapping in the pixel shader we calculate which segment of the ray falls between 0 and 1. On this segment we search for the intersection with the terrain. For this new method, the segment of the ray that enters below the triangle mesh usually starts far below 1. Therefore the segment is shorter, and the terrain is very near to the entry point. An example can be seen in Figure 4.

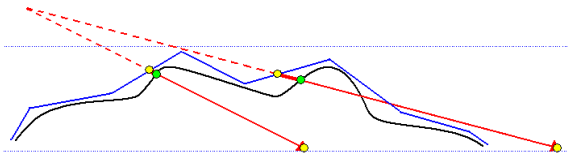

Figure 4: On approximate triangle mesh the segments of the rays are shorter and the terrain is nearer

The binary search or the secant method improved only slightly due to the shorter segment, but linear search became significantly faster. The search begins close to the intersection, generally the intersection is found within a few steps. Using 4 linear search steps, and 1 secant search step, the displayed terrain looks good (Figure 5).

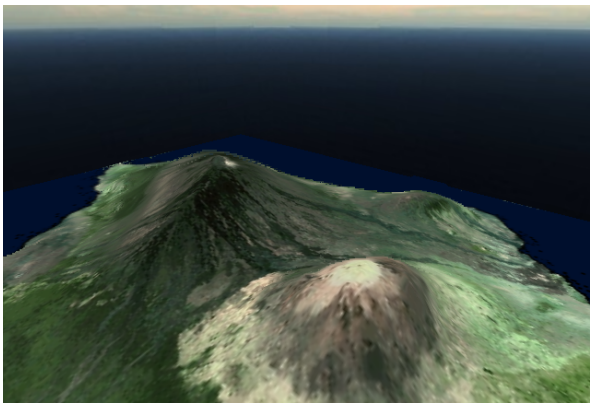

Figure 5: Terrain with 4 linear and 1 secant search steps.

For steep rays, the result is satisfactory, but when the camera is at low height, the upper part of the terrain

doesn't appear correctly. For near horizontal rays, the result is wrong. The triangles above the horizon show upside down, far away repeated terrain. Figure 6 shows the terrain rendered with basic linear search.

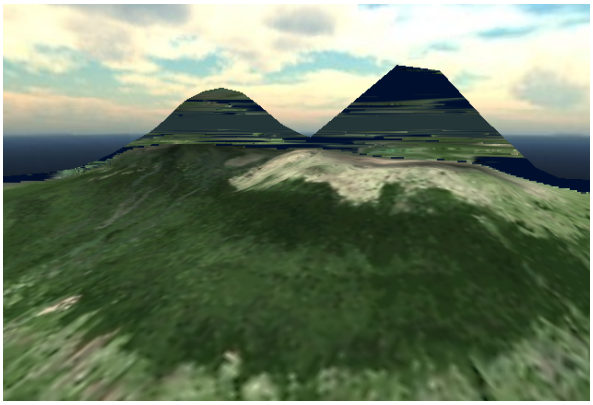

Figure 6: The problems of basic linear search.

4.1 Upward Rays

Basic displacement mapping assumes that the ray enters at $H = 1$ and exits at $H = 0$. The intersection will be found between these points. The ray never reaches $H = 0$ if it starts somewhere between 0 and 1, and is going upwards. The basic algorithm takes steps towards 0. On upward rays, it would also step towards 0. In this case these steps go behind the camera, because 0 can be reached only there. This is the reason for the appearance of the upside down terrain above the horizon. To solve this problem we should handle upward rays differently. It is sufficient to examine the ray's direction vector's H (3^{rd}, vertical) coordinate. It is positive for upward rays. The determination of the ray's endpoints needs to be changed. The endpoint of an upward ray is at $H = 1$ instead of $H = 0$. Results show that the above problem is solved by this change. Upward rays also find intersection (shown in Figure 7), but there are still problems with pixels (rays) close to the horizon.

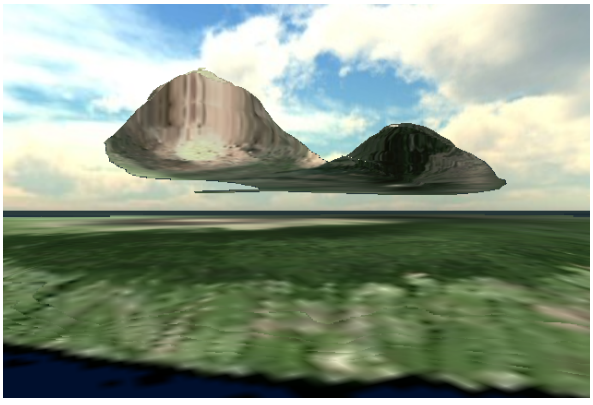

Figure 7: The problem of near horizontal rays.

4.2 Near Horizontal Rays

The original algorithm steps vertically the same ranges between the in and out points for linear search. When a ray is flat, a small vertical step could result in huge step forward on the ray. Figure 8 shows an example. It is possible that one step skips the whole terrain. In this case the intersection cannot be found. This is the reason why a part of the terrain doesn't appear in Figure 7. We found two different solutions for this problem: Exponential Search and Equidistant Linear Steps.

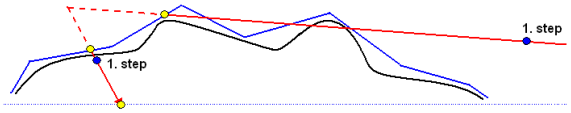

Figure 8: The problem of flat (near horizontal) rays.

4.3 Exponential Search

Standard linear search has the problem that for flat rays it uses too large steps and it steps over the whole terrain. A better solution is that the algorithm steps rather exponentially than linearly. At first the steps are very small, but increase exponentially. However, it reaches the end of the ray with finite steps, similarly to standard linear search. The range of the steps are:

$$H = 1 - 1/2^{N_{iter}-1-i} \tag{3}$$

where N_{iter} is the number of the iterations and i is iteration variable. H is the height of the steps on the ray segment. The steps go from the enter point – the H is 1 there, and go forward to the end of it – where H is 0. The current position is calculated by linear interpolation between the two tips by the following code:

```
uv = uv_in * H + uv_out * (1-H);
```

where `uv_in` the enter point and `uv_out` is the end point. Table 1 shows an example for exponential steps at 10 iterations.

The result is much better than for linear search (shown in Figure 9). Most of the horizontal rays find proper intersections, however not all of them. In a few pixel width strip there is still no intersection. The horizontal and almost horizontal rays are flat, they can have nearly infinite length. On these rays even a very small step skips the whole terrain (see Figure 8).

4.4 Equidistant Linear Steps

The main problem with the standard algorithm is that it steps vertically the same ranges to ensure it always reaches the end of the segment, but it does not consider the ray's

Iteration	Height of step
0	0.998046875
1	0.99609375
2	0.9921875
3	0.984375
4	0.96875
5	0.9375
6	0.875
7	0.75
8	0.5
9	0

Table 1: Exponential Search with 10 iterations

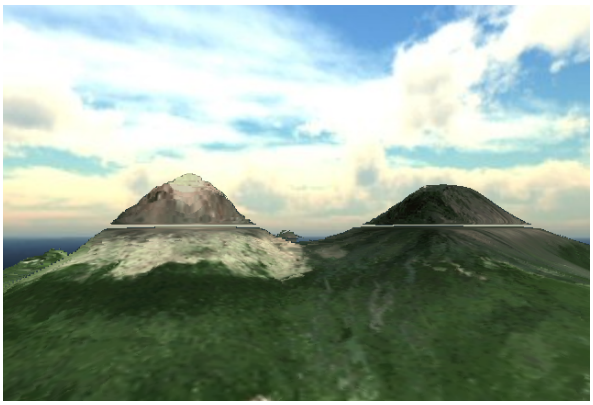

Figure 9: Terrain with Exponential Search, 10 iterations.

length. Therefore the lengths of the steps are not equal for rays of different declination angles.

Instead of stepping vertically, the algorithm should step equal distances. Thus, for every ray the length of the steps would be the same. At flat rays, the algorithm does not skip the terrain. Figure 10 shows the same example, but with the new linear search. It is observable that this time it does not step too much, and the intersection can be found. However a new problem appears. The former algorithms always can reach the end of the segment, but this algorithm can not. For example, if the iteration number is 5, than the upper ray in Figure 10 is unable to reach the real intersection.

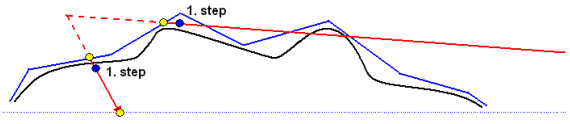

Figure 10: Improved linear search with fix step ranges.

To determine the step's range, at first the length of the ray's segment should be computed. Then a constant is divided by this length. At each step this value is subtracted from H[3].

$$\mathbf{t}_{view} = \frac{\mathbf{M}_{terrrain} * \mathbf{p}_{eye} - \mathbf{p}_{terrain}}{\|\mathbf{M}_{terrrain} * \mathbf{p}_{eye} - \mathbf{p}_{terrain}\|} \tag{4}$$

$$step = c\frac{\mathbf{t}_{view_z}}{(h_{end} - \mathbf{p}_{terrain_z})} \tag{5}$$

where $\mathbf{M}_{terrrain}$ is the UWH matrix, $\mathbf{p}_{eye}$ is the position of the camera and $\mathbf{p}_{terrain}$ is the UWH position of the pixel. h_{end} is 0 when the ray goes downward, 1 otherwise.

The constant depends on the resolution of the triangle mesh. If the resolution is high, then the real terrain is nearer, thus smaller steps give better results, so the constant is smaller. At lower resolution the distance between the triangle mesh and the real terrain could be larger, thus greater constant is better. E. g. at 32×32 resolution a good constant was 0.025, and 5 steps were enough.

For a small part of the pixels, this linear search could not reach the real terrain, thus the intersection cannot be found. Mostly this problem occurs for those rays which enter below the triangle mesh, but they are above the real terrain. To solve this problem, the algorithm handles these rays separately. If the intersection was not found, the search continues with more iteration. In the new iterations the steps could be large, e.g. twice as big as originally. As these rays are in minority, the algorithm would not be significantly slower. If the increased number of steps still cannot find an intersection (it is possible that they never would), then neither the pixel will be colored, nor the Z-buffer will be written.

Using standard linear search, the terrain will be striped. Therefore, the search continues after an intersection is found, but using secant search. The two start points of the secant search are the last two points of the linear search. With the penultimate linear step, the algorithm determines a point where the terrain is still below the ray. The last step shows that the terrain is above the last point. The surface of the real terrain intersects the ray between these two points. The secant method finds an accurate intersection quickly, thus the strips are eliminated.

As Equidistant Linear Steps perform much better than the Exponential Search, the Exponential Search is not used in the final version.

5 Results

It is hard to determine the optimal constant values in the algorithm such us triangle mesh resolution, first linear search's iteration number, second linear search's iteration number, step ranges, secant search's iteration number etc. The performance of the described algorithm is highly affected by the length of the ray sections which fall between the course triangle mesh and the real terrain. Average of these lengths depend on the local roughness and local curvature of the real terrain, but is nearly independent

[3] At beginning H is 1.

of the local steepness.Rendering speed also highly depend on graphics hardware. The algorithm was developed and tested on an Nvidia® Geforce® 8400M[4]. The viewport resolution was 640×480 pixels. The frame rate dropped at 64×64 quads, thus 32×32 quads was a better choice. With 5 linear steps most of the intersection was found. For pixels, where the search could not reach the intersection, 10 further linear steps followed. If the search was still inefficient, the pixel became transparent, thus other part of the terrain or the skybox could appear. However if the linear search entered below the real terrain, 2 secant search steps refined the result. Figure 11 shows the terrain with these values. The frame rate was between 55-100 fps[5].

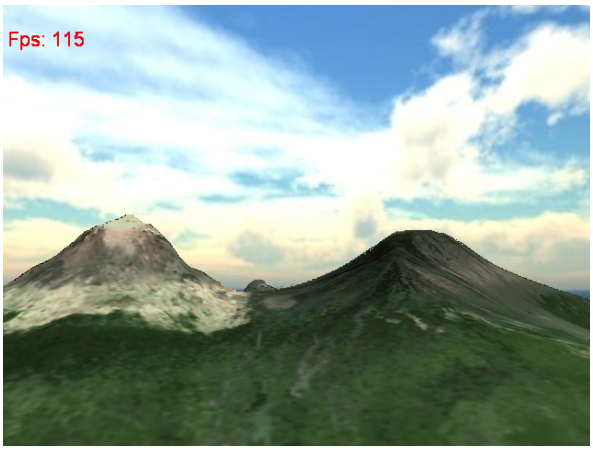

Figure 11: Final terrain.

As seen in Figure 11 the terrain appeared correctly. The algorithm finds the proper intersection for almost every pixel. However, at some pixels the algorithm makes mistakes. These mistaken pixels mostly appeared at the edge of hills.

Using the same paremeters, the frame rate on a desktop PC with Nvidia® Geforce® 8800 GT was much higher, over thousand fps. After changing parameter values, the algorithm was more accurate, pixel errors barely appeared. The triangle mesh resolution was set to 256×256, step ranges was the quarter of original, linear step numbers doubled. The result was still over hundreds of fps and with unnoticeable errors. Figure 12 shows an example, where the heightmap's and the texture's resolution was 2049×2049. Table 2 shows results on different GPUs and settings. As the rendering time was highly dependent of camera position, four points of view has been selected. It was intentional to use fundamentally different views (see Figure 13). Frame rate tests for different configurations were tested on these fixed views. Refresh times show minima and maxima of the four measured values.

[4]Notebook version

[5]Depends on percentage of terrain on screen

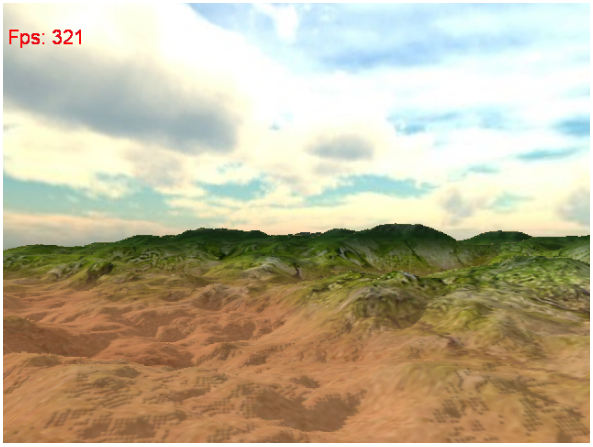

Figure 12: Final, 2049×2049 resolution terrain on 8800 GT with 256×256 quads.

GPU	Heightmap	Mesh	Speed[ms]
8400 M	257×257	32×32	9.52–18.87
8400 M	2049×2049	32×32	9.62–20
8400 M	2049×2049	256×256	22.22–41.67
8800 GT	257×257	32×32	0.55–0.8
8800 GT	2049×2049	32×32	0.55–0.83
8800 GT	2049×2049	256×256	3.7–3.82

Table 2: Rendering times

6 Conclusion and Future Work

There are several known algorithms that use only the pixel shader or only the vertex shader for displaying terrains. We developed a new technique that combines the pixel shader and vertex shader techniques. It resulted in a faster and more accurate algorithm than widely used ones. The main advantage of this new method is that it is independent of the heightmap's resolution. The approximate triangle mesh's resolution is constant, does not consider the heightmap's resolution and the displacement mapping is also independent of the heightmap's size. The frame rate is similar e.g. at 257×257 and at 2049×2049 resolution heightmaps.

However it is hard to determine the balance between speed and accuracy. If the decision is to be more accurate, the frame rate decreases. The method is also capable of reaching high frame rate on slower GPUs, but then accuracy has to be decreased.

In the future we shall make the algorithm better. E.g. improve the displacement mapping searches or make the triangle mesh dynamically changeable. A new method could be also promising: using height mipmaps the algorithm would not make mistakes, the proper intersection always could be found rapidly. Our research is going on by utilizing maps for local height minima and maxima. Instead of [2], the resolutions of these extremum maps are decreased in our research, similarly to the Mip-Map ap-

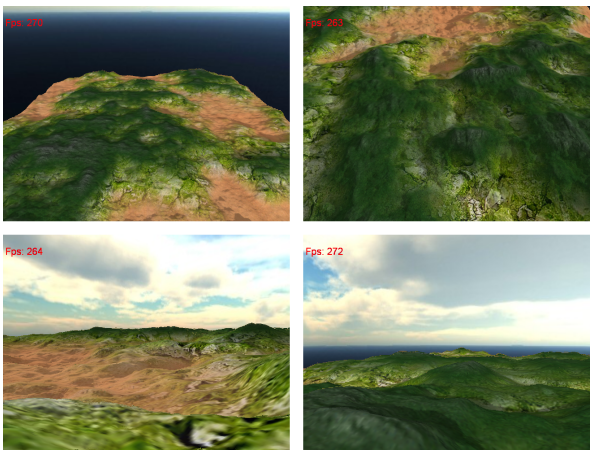

Figure 13: These four viewpoints were selected to be used during frame rate test of the algorithm on different platforms with different options. Shown frame rates were achieved on Geforce 8800 GT, 2049×2049 height map, 256×256 quads.

proach.

After a fast, accurate, detailed terrain displaying algorithm is developed, it could be improved by adding new elements, such as new detailed textures close to camera, dynamic light and shadows, vegetation, water etc.

Acknowledgement

This work has been supported by the Teratomo project of the National Office for Research and Technology, and OTKA K-719922 (Hungary).

References

[1] László Szirmay-Kalos, Tamás Umenhoffer. *Displacement Mapping on the GPU – State of the Art*. Computer Graphics Forum, 2008.

[2] Stephan Mantler, Stefan Jeschke. *Interactive landscape visualization using GPU ray casting*. Graphite, 2006.

[3] Christian Dick, Jens Krüger, Rüdiger Westermann. *GPU Ray-Casting for Scalable Terrain Rendering*. Eurographics, 2009.

[4] Renato Pajarola, Enrico Gobbetti. *Survey of semi–regular multiresolution models for interactive terrain rendering*. The Visual Computer 8: International Journay of Computer Graphics, pp. 583–605, 2007.

[5] Mark Duchaineau, Murray Wolinsky, David E. Sigeti, Mark C. Miller, Charles Aldrich, Mark B. Mineev-Weinstein. *ROAMing Terrain: Real-time Optimally Adapting Meshes*. IEEE Visualization, 1997.

[6] Jonathan Dummer. *Cone Step Mapping: An Iterative Ray–Heightfield Intersection Algorithm*. http://www.lonesock.net/files/ConeStepMapping.pdf, 2006.

[7] Barry Minor, Gordon Fossum, Van To. *Cell Broadband Engine Optimized Real–time Ray–caster*. 2005.

[8] Brian Smits, Peter Shirley, Michael M. Stark. *Direct Ray Tracing of Displacement Mapped Triangles*. Rendering Techniques 2000: 11th Eurographics Workshop on Rendering, 2000.

[9] Arul Asirvatham, Hugues Hoppe. *Terrain Rendering Using GPU-Based Geometry Clipmaps*. GPU Gems 2, pp. 27–46, 2005.

[10] Alex A. Pomeranz. *ROAM Using Surface Triangle Clusters (RUSTiC)*. Department of Computer Science, University of California, 2000.

[11] Jonathan Blow. *Terrain Rendering Research for Games*. SIGGRAPH 2000 Course 39, 2000.

[12] Peter Lindstrom, David Koller, William Ribarsky, Larry F. Hodges, Nick Faust, Gregory A. Turner. *Real–Time, Continuous Level of Detail Rendering of Height Fields*. Proceedings of SIGGRAPH 96, pp. 109-118, 1996.

[13] Stefan Rottger, Wolfgang Heidrich, Philipp Slusallek, Hans-Peter Seidel. *Real–Time Generation of Continuous Levels of Detail for Height Fields*. WSCG Proceedings 98, pp. 315-322, 1998.

[14] Jens Schneider, Rudiger Westermann. *GPU–Friendly High–Quality Terrain Rendering*. Journal of WSCG, 2006.

[15] Wikipedia. *Level of Detail*.