GPU-only Terrain Rendering for Walk-through

Sunyong Park
Soongsil University
Sangdo-dong
Dongjak-gu
156-743, Seoul, South Korea
aknyong@ssu.ac.kr

Kyoungsu Oh
Soongsil University
Sangdo-dong
Dongjak-gu
156-743, Seoul, South Korea
oks@ssu.ac.kr

ABSTRACT

Accurate terrain representation takes a very significant role in making a scene more realistic. In this paper, we present a full GPU-based real-time terrain rendering algorithm by ray-casting. Since it requires no geometrical structure like a polygonal mesh, it doesn't need any LOD (Level-Of-Detail) policies. Most of them are processed on CPU and may give much burden on the CPU. As a result, it enhances the whole performance of the system. Our method grants a complete freedom to the view point and its direction, so objects can move around so freely in the air or on the surface that it can be directly applied to any computer games and VR (Virtual Reality) system. To better the rendering quality, we applied curved patches to the height field. On the way, we suggest a simplification for evaluating a ray-patch intersection. We implemented all the processes on GPU, and obtained tens to hundreds of frame rates with a variety of resolutions of height maps: $256 \times 256 \sim 8192 \times 8192$ (texel^2).

Keywords

Height field · Ray-casting · PDM (Pyramidal Displacement Mapping) · Quad-tree · GPU-based Rendering

1. INTRODUCTION

In computer graphics, accurate representation of terrain plays an important role in making a scene more realistic. We cannot imagine any world without it at all. Its applications are, therefore, wide-ranged over many areas from GIS to virtual reality and computer games; None the less, terrain still remains a challenging area to game or VR creators. To represent or simulate terrain, we usually use the height map, which is also called the height field as a space. The height map is an image each pixel of which contains a height value of the corresponding position. There are two main approaches to construct-ing terrain from the height information.

The first is the polygon-based method [Gro95, Duc97, Paj98, Hwa04, Los04, and Asr05]. It generates a terrain mesh from a height map either in regular grids or adaptively to local complexity and map the corresponding height value to each vertex of the mesh. They are basically fast because they can utilize parallelized GPU capabilities. However, the number of polygons gets bigger as the complexity of terrain increases, so it may require too much time and storage. To relieve this inefficiency, LOD (Level Of Detail) is used on the basis of distance from view points and variation of heights. Even though it improves the performance of system, other serious problems are entailed between polygons of different levels, such as popping or cracking etc. Worse, most LOD algorithms are processed on CPU, and a great burden may be caused to the CPU.

The others are based on ray-tracing [Coh96, Ser97, Wri92, Lee95]. They render terrain just with a height map. More than one ray per pixel are made, cast, and traced into the height field until they intersect with the field. Once an intersection is found, the pixel is shaded using the positional information. They are based on a hierarchical data structure, and therefore less affected by geometrical complexity, whereas they are not fit to real-time applications.

In this paper, we present a real-time algorithm based on GPU ray-casting. Our method was much inspired by the Pyramidal Displacement Mapping by Oh et al [Oh06]. The PDM renders the height field using a pyramid of depth images. Here we use the term ***height*** inversely with ***depth*** (Figure.1). The image pyramid is a set of images that its width and height decrease by 1/2 as a level goes up, which has the original depth image at level 0. Each texel of sub-image has *the minimum depth* (or *maximum height*) of adjacent 4 texels of the immediate lower level. Finally, the highest level of the pyramid has just one texel whose value is the minimum of the entire depth map. We can efficiently and accurately find ray-

height field intersections by this data structure.
When a ray goes down, the ray can safely move down to the maximum height of one texel area without any intersections, and then the current height of the ray is read from the next lower level. The ray moves forward in the same way until it reaches level 0. If the height from the lowest level is equal to or higher than the current position of the ray, we can say that the intersection is found. There are, however, some problems if the PDM is applied to terrain as it is. As the displacement mapping handles down-displacement from the mapping plane, the view point always looks downwards (Figure.1-1). In terrain rendering, the view point can go under the mapping plane near the surface and look upwards. We gave some significant revision to the original PDM to cope with this situation. The ray traverses the same hierarchy as downward cases, but the way to trace a ray is slightly different. When the current texel area is lower than the current ray, there cannot be any intersections within the area, so the ray can safely go up to the boundary of the texel, not the maximum height this time. If the current texel in level 0 is higher than the ray, we get the intersection. If not, the ray keeps going forward until it finds an intersection. There occurs one more problem that the surface looks like staircases as the view point gets closer to the surface. To this, we reconstructed the surface with bilinear patches. One bilinear patch is built from four adjacent height elements, and a ray-patch intersection is evaluated there. We simplified the ray-patch intersection method given in [Ram04], and made it feasible for the pixel shader.
Our method is based on the idea that the ray has no intersections to the maximum height in downward directions, and to the texel boundary in upward directions. However, when we reconstruct the surface with bilinear patches, some part of one patch may be higher than the maximum height of the correspond-ing texel, where the ray may pass through the patch. To solve this problem, we devised a new depth map (we call it the bounding map) which has as a texel value the height of each bounding volume that encloses each bilinear patch. The ray traversal on this map is exactly the same as that of the original map.
Our algorithm runs fully on GPU, so it allows the CPU to be dedicated to other tasks, like physics-based rendering, resource managements and AI etc., so it enhances the whole system performance.
The major contributions of our work are as follows:

1. Overhaul the existing PDM algorithm to give the full freedom of view point and direction, allowing freely fly through and walk through terrain.
2. Dramatically improve the visual quality of the terrain surface by applying curved patches.
3. Enhance the system performance by carrying out the whole process on GPU.

The remainders of this paper consist of follows. Section 2 summarizes related works and section 3 overviews our system. Section 4 describes rendering terrain using a hierarchical data structure without curved patches. Section 5 applies curved patch to the terrain. Section 6 shows finally rendered results.

2. Related Work

So far, most terrain rendering techniques that provide sufficient freedom to the position of view point and its direction have been based on polygonal structure [Gro95, Duc97, Paj98, Hwa04, Los04, and Asr05]. Gene-rally, they render terrain in fashions that make terrain meshes and map textures to them. Those methods are well fit to the rendering pipeline of GPU, and therefore have some advantage on rendering time, whereas the mesh should be prepared in advance and the number of polygons should be controlled, otherwise it may impose a big load on the system. We usually use LOD (Level Of Detail) to handle this problem, whose purpose is to reduce sub-sampling artifacts. However, the problems accompanied like cracking and popping etc., since they should be processed on CPU, incur more inefficiency. Naturally main concerns in polygon-based methods have been focused on handling re-meshing due to the LOD.

There have been many researches from this point of view. Markus et al. introduced the adaptive quad-tree meshes for regular grids of terrain data using dyadic scaling of the wavelet transform [Gro95]. Pajarola suggested restricted quad-trees (RQT), which is an adaptive, hierarchical triangulation model and is used to triangulate a parametric surface [Paj98]. Duchineau et al. used preprocessed bin-tree triangles with view-dependent, guaranteed error metrics to do re-meshing in real-time (ROAM: Real-time Optimally Adapting Meshes) [Duc97]. Pomeranz presented RUSTiC (ROAMing Using Surface Triangle Clusters), which is the ROAM that every triangle bin should have the same boundaries on the shared edges in order not to have any cracks [Pom00]. Lok et al. replaced the triangle bin-tree with the diamond data structure. Their method uses an efficient out-of-core algorithm with GPU memory as

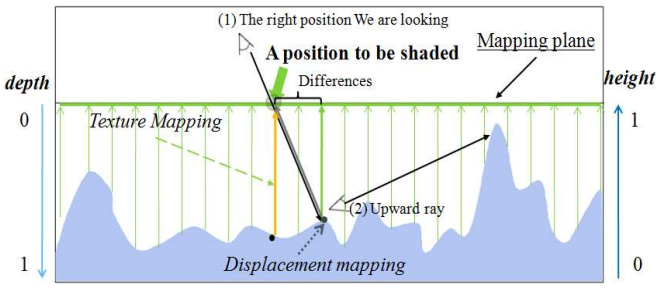

Figure 1. In the displacement mapping, the view point cannot go under the mapping plane, so there are just downward rays (1), but in terrain, upward rays are frequent (2).

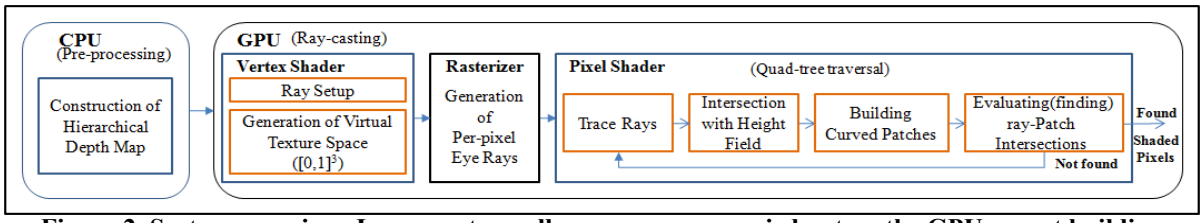

Figure 2. System overview: In our system, all processes are carried out on the GPU except building the hierarchical map, which is performed just once and doesn't have any influence on the system.

a cache. Hoppe et al. presented "the geometry clip-map" in [Los04] as a variant of texture clip-map [Tan98]. The geometry clip-map has an importance in that it processes the LOD on GPU. The clip-map caches terrain data in nested regular grids and it is refilled incrementally and toroidally as the viewpoint moves, where the vertices of grids are stored in a vertex buffer. However, it has a fixed grid resolution, and when a view point moves near the surface (like walk-through), bottlenecks may occur in updating clip-maps. In [Asr05], they upgraded the above clip-map to the GPU-only version using vertex texture.

Unlike polygon-based methods, ray-casting based ones directly cast and trace rays into height field, and find intersections. Cohen et al. [Coh96] introduced a CPU-based ray-casting algorithm. They used a voxel map made in regularly spaced heights from height map and top-down pyramid traversal. In [Coh96], they more developed this and implemented "visual fly-through" over vast amount of terrain data through the proper memory pre-fetching. Though this method was much advanced compared to other CPU-based algorithms [Ser97][Wri92][Lee95], still it was performed on CPU and just used for a fly-through purpose, not general.

In this paper, we propose a GPU-only algorithm that utilizes a hierarchical depth image. Several GPU-based techniques for displacement mapping have been introduced with a good performance and quality. Parallax occlusion mapping approximates ray-height field intersections with linearly interpolated parallax displacement [Tat06]. Relief mapping mitigated the artifacts using the binary search, but they have some problems in specific view directions and high frequency area respectively [Pol05]. Dynamic parallax occlusion mapping relieves the artifacts by varying sampling rates according to the ray's direction and frequency of the height field, but it just relieved the problem [Tat06]. Oh et al.'s Pyramidal Displacement Mapping solved above problem completely [Oh06] using a hierarchical data structure and a traversal algorithm fit to the hierarchy. Tevs et al. applied the similar scheme and used the bilinear patch for visual artifacts, but since they adopted the binary search to get intersections, it shows the same problem as [Pol05] in grazing angles [Tev08].

The method proposed in this paper is based on the PDM, which is a kind of displacement mapping and therefore is not fit to applying to terrain. We got rid of all these limitations. We are assured that this is the first case to apply GPU-based ray-casting to terrain.

3. System Overview

Our system consists of a preprocessing on CPU and a ray-tracing on GPU.

In the preprocessing step, a virtual space where a height field resides is constructed and the PDM of the height map is made. Ahead of that, the height map should be transformed into a depth map. We will use depths from *the mapping plane* (Figure 1) over the whole system. Since this step is performed just once and doesn't have much influence on the system.

In the ray-tracing step, per-pixel eye-rays are made and cast into the height field. In vertex shader, four rays connecting a view point and four corners of a screen-aligned rectangle are set up, and per-pixel rays are made through the rasterizer, the number of which is the same as the resolution of display. The pixel shader casts each ray into the virtual space and gets an intersection with the height field. The cast ray traverses the hierarchical map from top to leaf.

Figure 2 illustrates these two steps in more detail.

4. Terrain Rendering with a Depth Map

The Pyramidal Displacement Map (PDM)

The PDM is an image pyramid of depth maps whose sublevel pixel has the lowest value of quadrant pixels of the immediate superlevel. Since the original PDM is for displacement mapping, it has some constraints that viewpoints cannot go under the mapping plain and its direction always looks downwards (Figure 1). In this paper, we lifted all these constraints and enabled view points to go down around the surface and look toward any directions.

A Virtual Space and Per-pixel Rays

To render the height field, we first establish *a base space* in which a height field will be located and into which all related coordinates will be transformed. In Figure 3 (a) shows a virtual space defined within the world coordinate system, and finally the space is scaled to the 3D unit space $[0, 1]^3$ so as to be aligned with a height field (texture space). Since we represent *high-and-lows* of terrain by depth from *the mapping plane*, we use the coordinate system as described in Figure 3(c) for convenience. After the virtual space is set up, per-pixel eye-rays are generated using the

 ISBN 978-80-86943-93-0

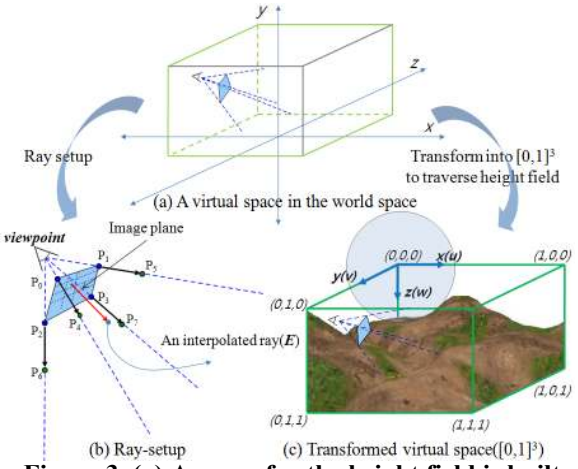

Figure 3. (a) A space for the height field is built. (b) In the vertex shader, four basic rays are set up, which are on lines connecting a viewpoint and four corners of image plane. Interpolating them, per-pixel rays (*E*) are generated, (c) All coordinates are transformed into the normalized virtual space [0, 1]^3 for the space to be aligned with the texture space. We have the z-axis turned-over as we use the depth map.

rendering pipeline. We make a viewport-aligned rectangle, which can be obtained by inversely transforming four points (1, 1), (1,−1), (−1, 1) and (−1,−1) of the projection space into the view space. In vertex shader, four basic rays connecting the view point(V_p) and four points P_0, P_1, P_2, and P_3 are made and, through rasterizer, per-pixel rays are obtained(Figure 3-b).

Ray Casting

Traversal algorithms adopted in [Ser97, Wri92, Lee95] are just related to downward rays since any view point cannot go under the mapping plain (Figure 3). However, because in case of terrain rendering, the view point can be located around the surface, naturally upward rays are generated. We made it possible for the ray to travel in any directions at arbitrary view points with some improvements.

4.1.1 Downward Ray

The hierarchical structure helps us find an intersection more quickly. When a ray descends, the ray can safely move to the maximum height or the texel boundary because the ray has no intersections in that area, and then the position is read from the next lower level. This process is repeated until the ray reaches level. If the depth read from level 0 is equal to or lower than the end point of the ray, we get the intersection. Figure 4(a) illustrates its general algorithm. After the ray arrives at the level, however, if it did not have an intersection, it searches the height field linearly. In the worst case, its time complexity is O(n). We improved the performance to O($\log_2$n). A ray crosses one grid, and if its position is still higher than the surface, the level of the PDM

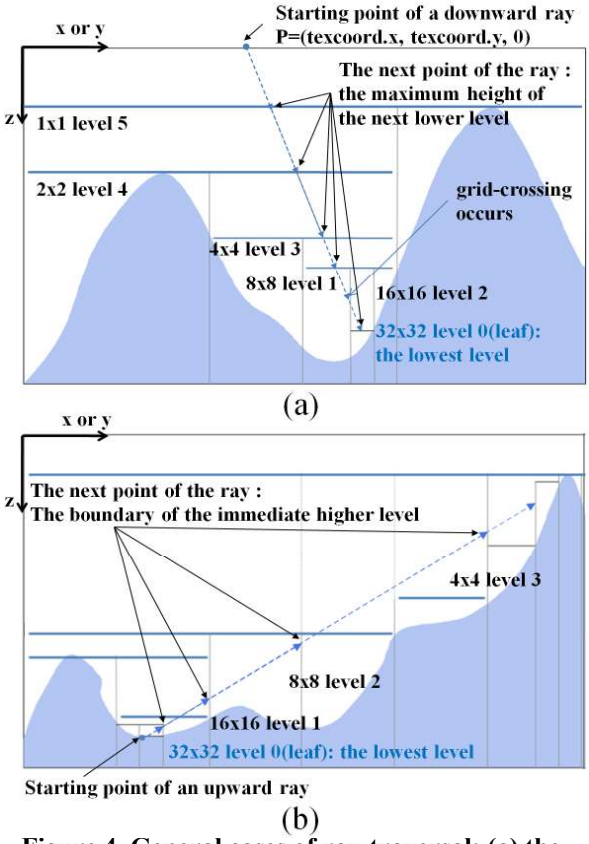

Figure 4. General cases of ray-traversal: (a) the ray can safely advance to the maximum height or the boundary of the current texel without intersections, (b), and the ray can advance to the boundary of the current texel.

raises one level up by force so that the ray skips twice as long distance as the previous level. We reduced the number of node-crossings greatly in this way.

4.1.2 Upward Ray

The ways a ray goes up is a bit different from downward cases. The ray moves just to the texel boundary, not the maximum height this time. If the current texel value in level 0 is larger than or equal to the height of the current ray, we have got an intersection. As a view point stays around the surface, more upward rays are generated and should be treated more efficiently. One straightforward method is to linearly search the height field. The linear search is very easy, but its cost increases proportionally to the resolution of height map. We take advantage of the quad-tree structure to reduce the cost (the number of advances). Nevertheless, when a view point is located in a deep valley, the linear search at the location of level 0 is inevitable. It usually takes up a considerable portion of the cost. We raised one level up each time a ray fails to find an intersection at each advance and thus improved the performance.

Rendering Results using PDM

Figure 5 shows results rendered with a depth map and above mentioned algorithms. As shown in the figure, when the terrain is flown through (a), it appears to be no problem, but as the view point approaches the surface (b), it looks like staircases. To alleviate this visual discomfort, we reconstruct the surface with curved patches(to be explained in the next section).

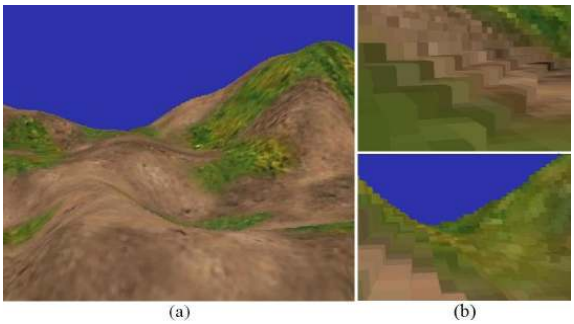

Figure 5. When we look at a height field from the far, it appears to be no problem (a), As we get closer, it shows some visual artifacts (b)

5. Terrain Rendering with a Depth Map

To improve the quality, we covered the height field with bilinear patches, considering computational efficiency and implementability in shader program.

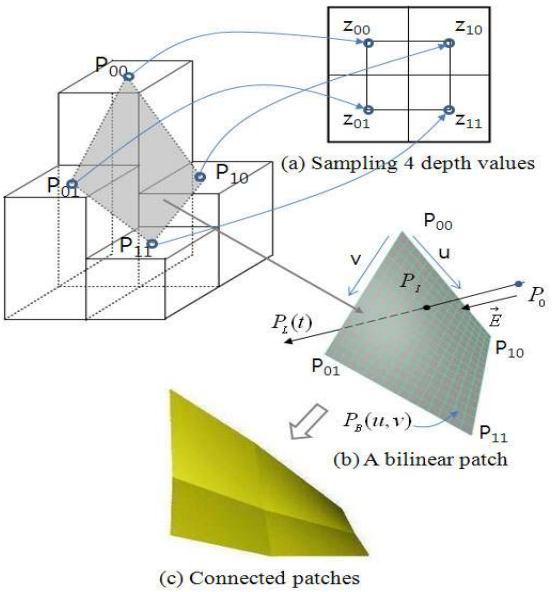

Figure 6. (a)A bilinear patch is built from the nearest four texels. Because the uv plane of height field is of uniform grids, we can make 3D coordinates by just sampling depths (b) A bilinear patch built (c) Four patches

Ray-Bilinear patch Intersection

A bilinear patch is built from the nearest four texels of the location where a ray hits the height field. Since the height field consists of uniform square grids, we can immediately know x, y coordinates, and therefore we just need four depth values (*z*'s) to complete full 3D coordinates(x, y, z). Figure 6 illustrates this and a bilinear patch made from the process. A ray-patch intersection can be evaluated mathematically by parametric equations shown in Figure 7. While this procedure was hinted by [Ram04], we simplified it significantly. Generally we need three quadratic equations with two parametric variables to express a bilinear surface in 3D space, one equation for each dimension, which can be written $P_B(u, v)= (P_B(u,v)_{.x}, P_B(u,v)_{.y}, P_B(u,v)_{.z})$. In the same way, a line is $P_L(t)= (P_{0.x} + t \cdot E_{.x}, P_{0.y} + t \cdot E_{.y}, P_{0.z} + t \cdot E_{.z})$. As a result, to get an intersection of a curved surface with a ray, we get three variable quadratic equations with respect to u, v, and t. We simplified these into one variable quadric equation, which come from substitution of two linear equations into a quadratic equation, $(u, v, P_B(u, v)_{.z}) = (P_{0.x} + t \cdot E_{.x}, P_{0.y} + t \cdot E_{.y}, P_{0.z} + t \cdot E_{.z})$.

[1]However, the proper solution should be inside the bilinear patch. Figure 12 shows how to discriminate it from two real solutions of the equation (1). Any rays first start on a boundary of ***the bounding volume of a texel,*** where $t = 0$. In the concave case (a), if the smaller *t is less than 0*, since it is located behind the patch, the boundary is performed for the other larger *t*. If the larger *t* is inside the patch, that's the proper solution, otherwise, the ray moves forward more until it finds an in-patch intersection with the next patch (c). In the convex case (b), simply the smaller *t* is the appropriate solution.

1) Bilinear patch

$P_B(u,v)$

$=(1-u)\{P_{00}(1-v)+_{01}v+P_{01}v\}+u\{P_{10}(1-v\}+P_{11}v\}$

$=(u, v, (1-u)(1-v)z_{00}+v(1-u)z_{01}+u(1-v)z_{10}+uvz_{00})$

,where $0 \le u, v \le 1$.

2) Ray

$P_L(t) = P_0+t\mathbf{E} = (P_{0.}x+t\mathbf{E}.x, P_{0.}y+t\mathbf{E}.y, P_{0.}z+t\mathbf{E}.z)$

3) Ray-Bilinear Patch Intersection

$P_L(t) = P_B(u,v)$

$\Leftrightarrow (P_{0.}x+t\mathbf{E}.x, P_{0.}y+t\mathbf{E}.y, P_{0.}z+t\mathbf{E}.z)$

$= (u, v, (1-u)(1-v)z_{00}+v(1-u)z_{01}+u(1-v)z_{10}+uvz_{00})$

$\Leftrightarrow u = P_{0.}x+t\mathbf{E}.x$

$v = P_{0.}y+t\mathbf{E}.y$

$t = \{uv(z_{11}-z_{10}-z_{01}+z_{00})+u(z_{10}-z_{00})+v(z_{01}-z_{00})+z_{00}-P_{0.}z)/\mathbf{E}.z$

$= (uvA+uB+vC+D-P_{0.}z)/\mathbf{E}.z$

, where $A= z_{11}-z_{10}-z_{01}+z_{00}$,

$B= z_{10}-z_{00}$, $C= z_{01}-z_{00}$, $D= z_{00}$.

$\Leftrightarrow \mathbf{E}.x \cdot \mathbf{E}.y \cdot At^2$

$+\{(\mathbf{E}.x \cdot P.y+P.x \cdot \mathbf{E}.y)A+\mathbf{E}.x \cdot B+\mathbf{E}.y \cdot C-\mathbf{E}.z)t$

$+(P.x \cdot P.y \cdot A)+P.x \cdot B+P.y \cdot C+D-P_{0.}z)=0$ ·····equ.(1)

Figure 7. A simple ray-patch intersection

[1] This paragraph will be easy to understand after reading the following sections on the 'bounding map'

Challenges to Finding Intersections

To trigger building a bilinear patch, a ray should first meet the height field. Figure 8 illustrates three cases hard to get ray-patch intersections by the method of the previous section: (1) the ray meets the height field but doesn't meet with the patch. In this case, the ray advances more grids one by one, and then finds a valid intersection with a patch. (2) The ray has an intersection with a patch, but because it doesn't meet the height field, it passes through the right patch, (3) the ray is very closely related to the ray which passes

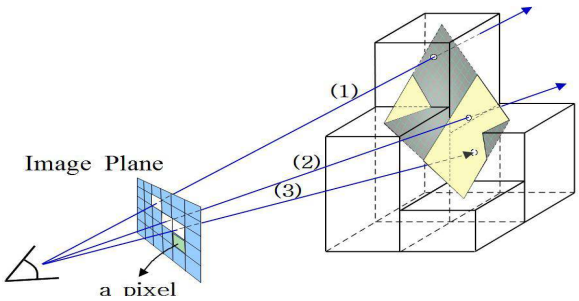

Figure 8. Three cases hard to find ray-patch intersections. (1) The ray meets with the height field but not with the patch. (2) It doesn't even build a patch since it doesn't meet the height field. (3) The ray should have already intersected with the previous patch. If the case (2) is properly handled, it cannot occur.

through the right patch. If (2) is properly handled, it never occurs. The last two cases make jaggies or holes in the middle or at the silhouette of terrain.

The Bounding Map

To keep a ray from passing through the right patch, we use a new depth map. Figure 9 shows how to build the new map. As shown, the new map forms a

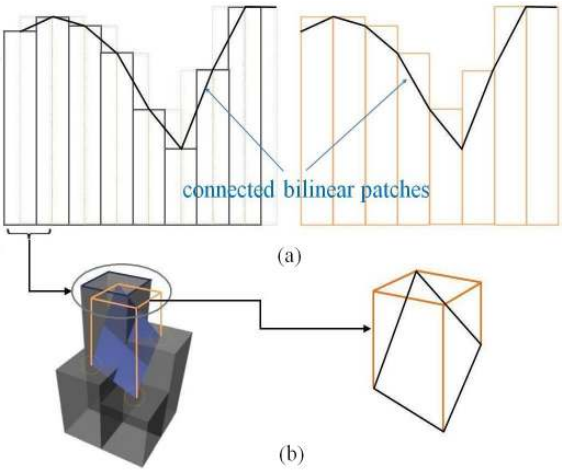

Figure 9. The Bounding Map. (a) The new map becomes a big bounding volume of the whole height field(we call it the bounding map) (b) One bounding volume enclosing a bilinear patch

big bounding volume of the entire height field (Figure9-b), so we call it the 'bounding map', which consists of texel-sized bounding volumes (Figure 9-c). It can be noticed that any ray with an intersection should first meet a bounding volume (Figure 9-b).

To build the bounding map, we pick the maximum height every four adjacent depth out and write it on the corresponding position of the new map. This bounding map is made into a pyramid. The original map keeps its original shape and is used to build the bilinear patch and find intersections by positional information handed over from the bounding map.

Ray-Casting and Ray-Patch Intersection in different maps

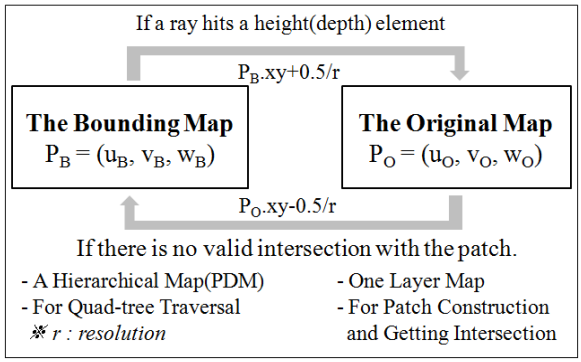

Figure 10. Quad-tree traversal and Bi-linear patching on the two maps

We evaluate a ray-patch intersection on two different maps with two separate processes. Firstly, a ray is cast into *the bounding map*. If *an intersection with the height map* exists, its position is handed over to the original map. And then, *a ray-patch intersection* is calculated. If it hits, the proper pixel is shaded. If not, the ray gets back to the bounding map and keeps to traverse until it finds an intersection (Figure 11-c, d). Figure 10 shows interactions between two maps.

Final results with Bilinear-Patching

Figure 11 shows rendering results without and with bilinear-patches. We can see a staircases-like surface in (a). After the surface is bilinear-patched, which we shaded it by point sampling to clearly see the shape of bilinear patches, those artifacts almost disappeared (b). (c) is the final scene that the bi-linear sampling is applied, where its quality is greatly improved.

Figure 11. Results (a) without patches (b) bilinear patching + point sampling (c) bilinear patching + bi-linear filtering

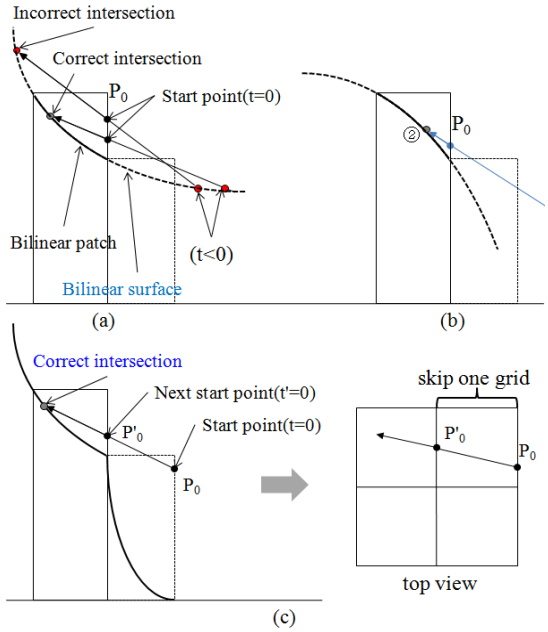

Figure 12. The ray always starts on the boundary of a bounding volume (t=0). There is two types of intersection either with a convex side (a) or a concave side of the patch (b). (a) Each t ($\geq$0) is checked if any intersection is made inside the patch. If not, the intersection occurs beyond the patch, so (c) the ray advances more until it finds an in-patch intersection with the next patch. (b) The real smaller t is the proper solution.

6. Experiments and Results

We implemented our method with DirectX 9.0 on ATI 2900 graphic card and 2.33Ghz Intel CPU.

[Table 1] compares our method to a polygonal one, where each terrain mesh has twice the number of triangles than that of height map. If the meshy terrain is rendered without an [2]LOD/ VFC, the performance is sharply down at the resolution of 1024^2 and at the higher resolution, memory overflows since the number of triangles exceeds the [3]maximum primitive count of hardware. Even though it applies an LOD/ VFC, the same problems ultimately occur at higher resolutions. On the contrary, our method, due to its hierarchical structure, is less affected by the increase of resolution and causes no memory overflows to the maximum resolution the hardware supports.

Figure 13 visualizes the number of node-crossings through spectrum. When the view point flies through (Figure.13-d), 90% of intersections are found within 16 (blue, sky-blue). Figure (b), (c), which are viewed on the surface, show more than 85~90% are done in 35. Just around 5% are more than 85, which occurs in grazing angles. Figure 14 shows rendering results carried out with various height maps.

Res.	Our method		[4]Polygonal method	
	PreProc (sec)	Fly/ Walk (fps)	NO LOD (fps)	LOD/VFC (fps)
256^2	0.6	105~115/ 66~99	250~285	290~340
512^2	1.8	53~75/ 47~56	75~83	180~230
1024^2	4.0	40~44/ 37~40	**19~22**	75~80
2048^2	8.1	58~98/ 54~85	overflow	**37~41**
4096^2	17.5	40~82/ 37~80	overflow	6~14
8192^2	36.0	32~67/ 30~45	overflow	overflow

Table 1. Time Performance-Resolutions

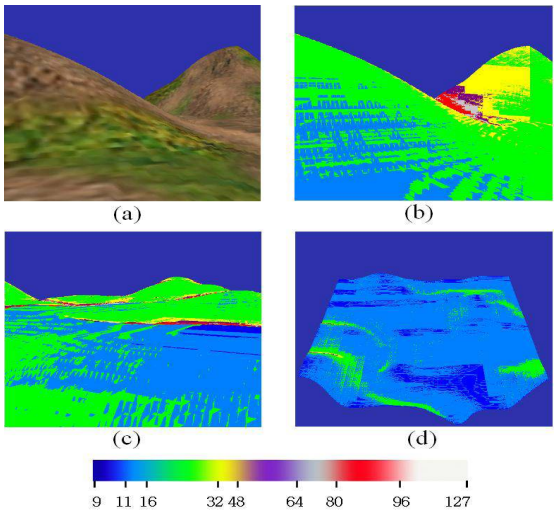

Figure 13. Spectrums representing the number of node-crossing (a) reference image, (b) and (c) walk-through, (d) fly-through

7. Conclusions

Thus far, we have presented a ray-casting based GPU-only terrain rendering method. We got rid of restrictions of displace mapping on the viewpoint and direction so that the viewpoint could freely move anywhere, and the problem that the surface looks like staircases near the ground was improved by applying bilinear patches. Our approach can be used in various fields, such as computer game, virtual reality, and flight simulation etc. In addition to that, our method will be better-suited for the ray-tracing based graphics hardware that will emerge in the near future.

8. ACKNOWLEDGMENTS

This research was supported by the Ministry of Culture, Sports, and Tourism, Korea, under the CTRC (Cultural Technology Research Center) support program supervised by the KOCCA and was also supported by Seoul R&BD Program (10581).

[2]LOD : Level Of Detail/ VFC : View Frustrum Culling

[3] The ATI 2900 Graphics card has the maximum primitive count of 8,388,607.

[4]The test was done on OGRE 3D using the octree for the spatial partition and with MaxMipMapLevel=5 for LOD.

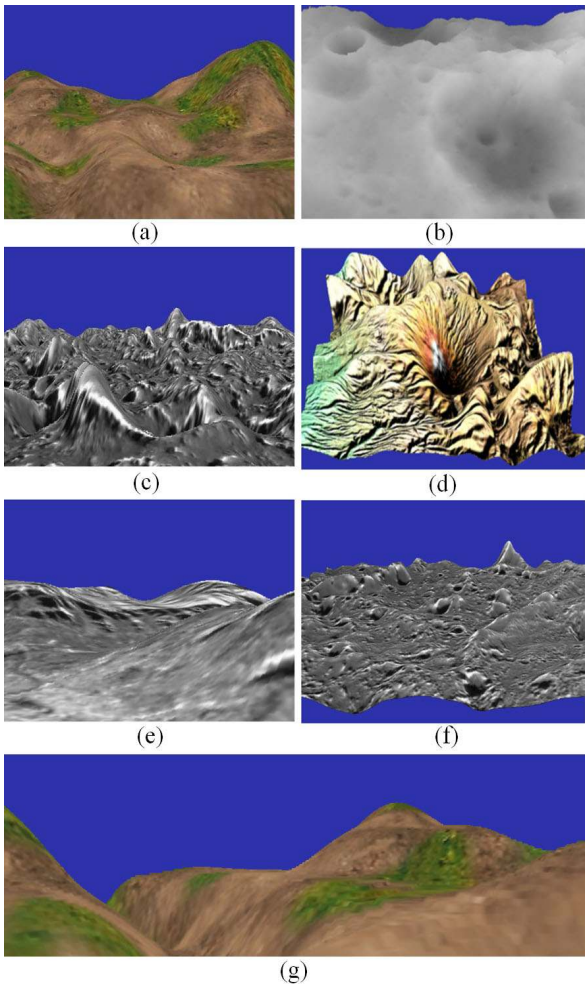

Figure 14. Final results rendered with a made height map (a), (g) and arbitrary height maps obtained from the web (b), (c), (d), (e), (f).

9. REFERENCES

[Asr05] Asirvatham, A., Hoppe, H.: Terrain Rendering using GPU Based Geometry clipmaps. GPU Gems 2 (2005)

[Coh96] Cohen-Or, D., Rich, E., Lerner, U., Shenkar, V.: A real-time photo-realistic visual flythrough. IEEE Transactions on Visualization and Computer Graphics 2(3), 255–265 (1996)

[Duc97] Duchaineau, M., Wolinsky, M., Sigeti, D., Miller, M., Aldrich, C., Mineev-Weinstein, M.: Roaming terrain: Real-time optimally adapting meshes. vis 00, 81 (1997)

[Gro95] Markus H. Gross, Roger Gatti, and Oliver G.: Fast multiresolution surface meshing. vis 0, 135 (1995)

[Hwa04] Hwa, L.M., Duchaineau, M.A., Joy, K.I.: Adaptive 4-8 texture hierarchies. In: VIS '04: Proceedings of the conference on Visualization '04, pp. 219–226 (2004)

[Lee95] Lee, C.H., Shin, Y.G.: An efficient ray tracing method for terrain rendering. In: Proceedings of International Pacific Graphics'95, pp. 180–193 (1995)

[Los04] Losasso, F., Hoppe, H.: Geometry clipmaps: terrain rendering using nested regular grids. In: SIGGRAPH '04: ACM SIGGRAPH 2004 Papers, pp. 769–776 (2004)

[Oh06] Oh, K., LEE, C., Ki, H.: Pyramidal displacement mapping : A gpu based artifacts-free ray tracing through an image pyramid. In: Proceedings of the ACM Symposium on Virtual Reality Software and Technilogy(VRST'06), pp. 75–82 (2006)

[Paj98] Pajarola, R.: Large scale terrain visualization using the restricted quadtree triangulation. vis 00, (1998)

[Pol05] Policarpo, F., Oliveira, M.M., Comba, L.D.: Real-time relief mapping on arbitrary polygonal surfaces. In: I3D ′05: Proceedings of the 2005 symposium on Interactive 3D graphics and games, pp. 155–162 (2005)

[Pom00] Pomeranz: Roam using surface triangle clusters(rustic). Master's thesis, California University (2000)

[Ram04] Ramsey, S.D., Potter, K., Hansen, C.: Ray bilinear patch intersections. journal of graphics tools 9(3), 41–47 (2004)

[Ser97] Sergei I. Vyatkin, Boris S. Dolgovesov, Valerie V. Ovechkin, Sergei E. Chizhik, Nail R. Kaipov "Photorealistic imaging of digital terrains, freeforms and thematic textures in real-time visualization system Voxel-Volumes", GraphiC-on '97, Moscow (1997)

[Tan98] Tanner, C.C., Migdal, C.J., Jones, M.T.: The clipmap: a virtual mipmap. In: SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques, pp. 151–158 (1998)

[Tat06] Tatarchuk, N.: Dynamic parallax occlusion mapping with approximate soft shadows. In: I3D '06: Proceedings of the 2006 symposium on Interactive 3D graphics and games, pp. 63–69 (2006)

[Tev08] Tevs, A., Ihrke, I., Seidel, H.P.: Maximum mipmaps for fast, accurate, and scalable dynamic height field rendering. In: SI3D, pp. 183–190 (2008)

[Wri92] Wright, J.R., Hsieh, J.C.L.: A voxel-based, forward projection algorithm for rendering surface and volumetric data. In: VIS '92: Proceedings of the 3rd conference on Visualization '92, pp. 340–348 (1992)