Pyramidal Displacement Mapping: A GPU based Artifacts-Free Ray Tracing through an Image Pyramid

Kyoungsu Oh[†] Hyunwoo Ki[†] Cheol-Hi Lee[‡]

Soongsil University[†], Seoul University of Venture and Information[‡]

{oks, kih}@ssu.ac.kr[†], chlee@suv.ac.kr[‡]

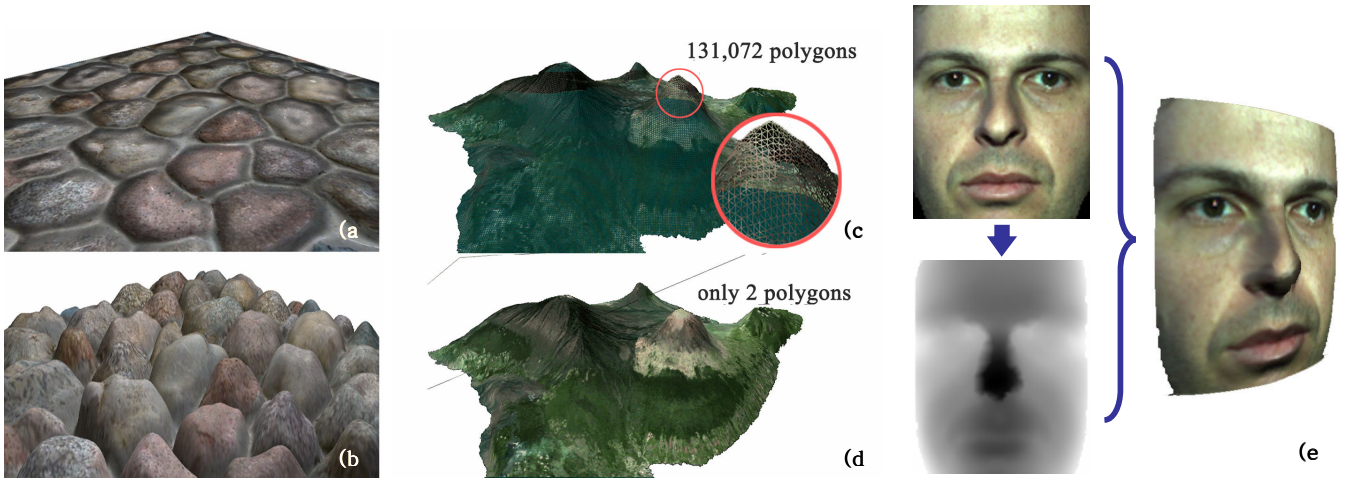

Figure 1. Our *pyramidal displacement mapping* (b, d, e) represents surface details with more realistic elevations than the *bump mapping* (a), and our method only needs a smaller number of polygons for rendering terrains than the *height mapping* (c). Our method also can use for rendering 3D images from a 2D image (e).

ABSTRACT

Displacement mapping enables us to details to polygonal meshes. We present a real-time artifacts-free inverse displacement mapping method using per-pixel ray tracing through an image pyramid on the GPU. In each pixel, we make a ray and trace the ray through a displacement map to find an intersection. To skip empty-regions safely, we use the quad-tree image pyramid of a displacement map in top-down order. For magnification we estimate an intersection between a ray and a bi-linearly interpolated displacement. For minification we perform a mipmap-like prefiltering to improve quality of result images and rendering performance. Results show that our method produces correct images even at steep grazing angles. Rendering speed of test scenes were over hundreds of frames per second and less influence to the resolution of the map. Our method is simple enough to add to existing virtual reality systems easily.

This paper is revised by Hyunwoo Ki (07-27-2007).

Additional information and example code are provided at his homepage, http://ki-h.com/.

Categories and Subject Descriptors

I.3.7 [**Computer Graphics**]: Three-Dimensional Graphics and Realism – *Ray tracing; Color, shading, shadowing and texture*.

General Terms

Algorithms

Keywords

Displacement Mapping, Real-time Rendering, Image-based Rendering, GPU, Quad-tree

1. INTRODUCTION

In virtual reality systems and video games, texture mapping is a significant method. It can present complex appearance without increasing geometry data by mapping images to polygons. Bump mapping [1] is one of the texture mapping methods that maps height values to polygons. Displacement mapping [2] is a method that does not only shade, but also moves geometry data according to height values. A displacement means a distance from a surface to a height-field, to be rendered onto a screen (see Figure 2). Inverse displacement mapping [3] determines an intersection between a ray and a height value for shading without changing geometry data (see Figure 3).

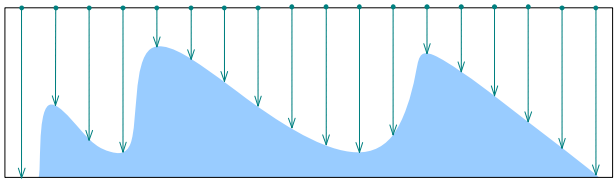

Figure 2. The example of a displacement map. Each texel of the map stores the *displacement* from a surface to an actual height.

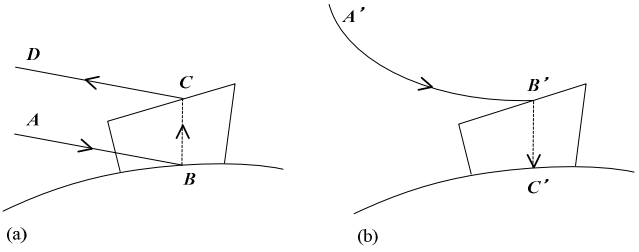

Figure 3. Concepts of displacement mapping (a) and inverse displacement mapping (b) methods (redrawn from [3])

Recently proposed techniques that exploit modern graphics hardwares for finding ray/height-field intersection, such as parallax occlusion mapping [10], relief mapping [11] are able to render complex scenes in real-time. They, however, cause visually serious artifacts since they approximate ray/height-field intersections using linear search or (and) binary search to (see Figure 5, 19 and 23). To avoid this problem, there are some adaptive sampling techniques such as [11, 12], but the error remains on a sharp height-field and at steep grazing angles. There are pre-processing methods for pre-computing safety radius of each path step of a ray [13, 14], but the pre-processing cannot be performed in real-time. They induce worsening of speed for pre-processing or need high dimensional texture.

We can use inverse displacement mapping for rendering terrains represented by height-field data. Previous ray tracing methods onto height-field often run on the CPU [4, 5, 6, 7], but these approaches take high-cost for rendering. A method uses an image pyramid of displacement data [4] builds a quad-tree [15] from the height-field. It applies incremental algorithm [8, 9] to a ray casting using the pyramid data for speeding up performance; however this method also cannot run at interactive rates, moreover it use super sampling for magnification filtering of the displacement data.

We present a novel per-pixel ray tracing technique through the pyramid of a displacement map on programmable graphics hardware. The pyramid data, called the *pyramidal displacement map*, is a quad-tree image pyramid which is a collection of images of reduced resolution from the most detailed displacement map (2^n by 2^n) to a 1 by 1 map. Each pixel of a sub-level image stores the minimum displacement value (i.e., maximum height value) of corresponding four cells in the below level. At rendering time, we create a ray for each pixel. The start position of the ray is the interpolated texture coordinate for the pixel and the direction vector is viewing vector transformed to tangent space. We trace the ray from the current texture coordinates to a point where the ray is lower than a stored displacement value. Our method can efficiently skip empty space by using the pyramid data (see Figure 5). Figure 1, 4, 19, and 23 show the power of our method.

Our method can render without aliasing artifacts even at grazing view angles. Rendering speed is less influence of the resolution of the pyramid data. We also introduce a bilinear interpolation filtering for magnification, and present a LOD technique for minification.

Section 2 discusses realted works. Section 3 describes our algorithm in details. Section 4 shows some results, and section 5 discusses conclusion with future work.

2. RELATED WORK

Cohen *et al*. [4] presented an accelerated height-field rendering algorithm using the CPU based *midpoint* technique. It uses a few additions, shifts and comparisons with a pyramidal data structure. The pyramid data structure is employed as a hierarchy of bounding boxes, where the value of each cell is defined by the maximum displacement. The method is faster than other CPU based methods [5, 6, 7] but it cannot be performed in real-time. In addition, it is difficult to present smooth elevations of terrains well.

Some GPU based height-field rendering methods [16, 10, 11, 12, 13, 14] can run at interactive rates. Parallax mapping [16] approximates surface *parallax* using displacement data.

Parallax occlusion mapping [10] approximates ray/height-field intersection with linearly interpolated parallax displacement. Such linear method, however, causes distracting errors on high-frequency displacement data. Relief mapping [11] reduces the artifacts by using a binary search technique. The binary search has a limitation by precision of a displacement texture. Besides, it produces searching errors if we do not increase sampling rates at steep grazing angles. Dynamic parallax occlusion mapping [12] relieves the artifacts by varying sampling rates according to a ray's direction and frequency of a height-filed, but it is not a complete solution for rendering artifacts-free images.

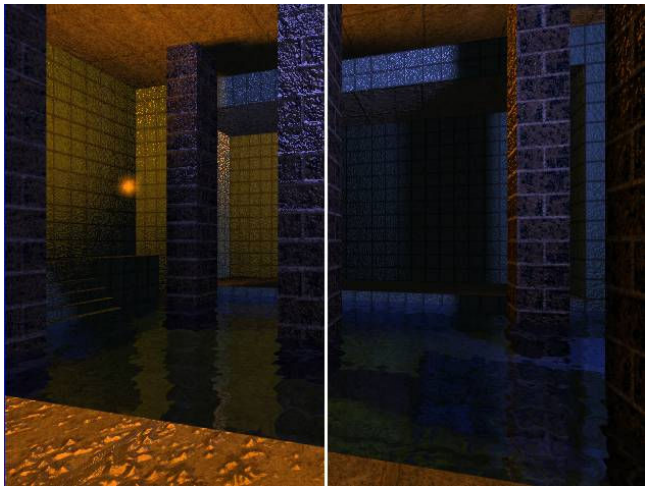

Figure 4. Addition of geometric details to improve reality of a virtual world (left: our method, right: bump mapping).

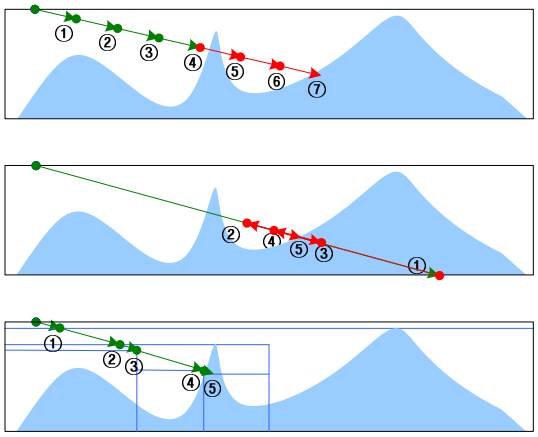

Figure 5. Linear search misses some intersections in particular at grazing angles (top), and binary search cannot be a complete solution (middle). Our method can avoid the artifacts (bottom).

Donnelly presented an accurate height-field rendering method using a volumetric texture that has pre-computed *safety distances* of rays [13]. Baboud *et al*. also used a pre-computed texture but they stored safety distance values into a 2D texture [14]. Such pre-computation processes require volumetric texture or a quite long preprocessing time. Thus, they are incongruent for fully interactive systems with users' interaction (e.g., depth acquisition of a user [18] and synthesis with a final image). Moreover, the time-cost for pre-processing and rendering increases as the resolution of a displacement texture is increased.

3. ALGORITHM

Our pyramidal displacement mapping uses a pyramidal data structure called *the pyramid of displacement map*. The pyramidal data structure represents the quad-tree of the displacement map. We traverse the quad-tree on the GPU for finding ray/height-field intersections rapidly and correctly. For this processing, we exploit dynamic branching and LOD texture fetching abilities of new graphics hardware. Pyramidal displacement mapping is a per-pixel lighting technique like previous GPU based displacement mapping methods, such as parallax occlusion mapping [10], relief mapping [11], etc [16, 12, 13, 14]. Our method can avoid serious problems (figure 4) that appear other recently proposed approximation techniques [10, 11, 12, 16] (see Figure 5 and 19).

An image pyramid is a collection of images of reduced resolution of the original image, from level 0 that has 2^n by 2^n pixels to level n that has 2^0 by 2^0 pixels. In an alpha component of each pixel (i, j) of sub-level texture, we store the smallest displacement value of the four pixels $(2i, 2j)$, $(2i, 2j+1)$, $(2i+1, 2j)$, $(2i+1, 2j+1)$ in the level below it (see Figure 4). Our algorithm performes top-down traversal of the quadtree.

As previous GPU based height-field rendering methods, such as parallax occlusion mapping [10], relief mapping [11], each pixel search the intersection with the height field as described below. First, we compute an eye vector in tangent space (E_{ts}). Eye vector is a direction vector from the eye to the pixel. We use this vector as direction of a ray. We set x and y values of the starting point of the ray to the current pixel's texture coordinates, and z value to zero. We project the ray onto XY plane of the pyramid texture of the current level. Then we advance the ray according to displacement sampled from the pyramid at x and y coordinates of the current ray. If the ray is above the height fields, then we advance the ray to that height and step down the image pyramid in one level; otherwise we step down the level without advancing the ray. We repeat these steps until the current level equals to a leaf level (figure 8). Finally, we compute illumination with normal and color at the end point of the ray.

Sub-section 3.1 describes how to create the pyramid of a displacement map, and subsection 3.2 describes an algorithm for finding accurate ray/height-field intersection using the pyramidal displacement map. We discuss a magnification filtering issue for rendering of interpolated height-field in subsection 3.3, and a minification filtering issue in subsection 3.4.

3.1 The Pyramid of a Displacement Map

First, we should create a pyramidal displacement map. We store the displacement value in alpha channel and normal value in RGB channels. We determine alpha value of each pixel of each sub-level texture with similar method to mip-mapping (*i.e.*, bottom-up approach). The pyramidal displacement map is an image pyramid which is a collection of images of reduced resolution from the most detailed displacement map (2^n by 2^n) to a 1 by 1 image. An alpha value of a pixel in current sub-level is the smallest displacement value (*i.e.*, the highest height value) of four corresponding cells (quadrant) in the below level (see Figure 6). Normal value is the average of four pixels in below level. This computation can be performed on both CPU and GPU. In PCI Express graphics hardwares, we can create the map quickly even if we perform it on the CPU (It costs about 2.3ms on a NVIDIA GeForce 7800 GTX 512MB graphics card). Besides, this process only executes when the displacement map is changed.

3.2 Ray/height-field Intersection

At each pixel, we compute a ray/height-field intersection using the pyramid of displacement map. We perform traversal of the tree from the root to the leaf (see Figure 7 and 8). As the aforementioned, a ray starts from an interpolated texture coordinates at each pixel. The z value of the ray is set to zero. The ray's direction is computed by converting eye vector into tangent space. The ray incrementally advances using the displacement value sampled from the current level's pyramid map at the ray's x and y coordinate.

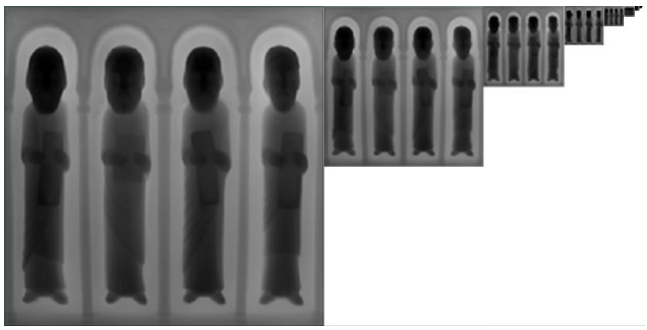

Figure 6. The smallest displacement value of four pixels in below level is the current sub-level's texel value.

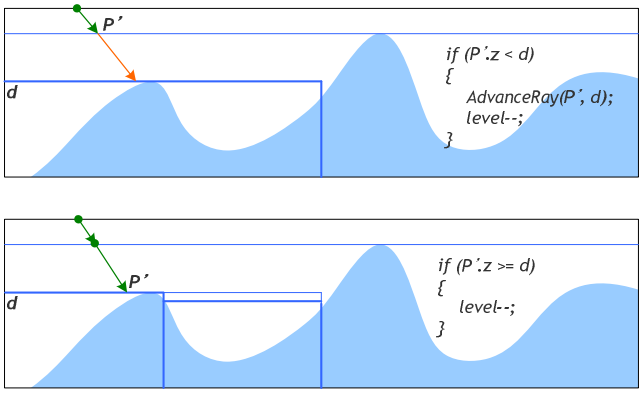

Figure 7. Ray tracing by comparison between z value at the current position of a ray ($P'.z$) and the displacement (d) at the pyramid of the current level

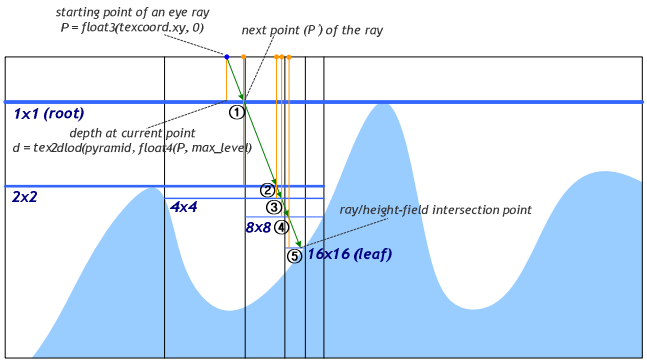

Figure 8. An example of a ray/height-field intersection using an image pyramid. A ray starts from the texture coordiniate of the current pixel (blue point). The ray advances from a root (1) to a leaf (5) according to displacement values sampled from the current level's pyramid map at the current ray's x and y position.

Main idea is as follows: Each pixel of image pyramid has a displacement value corresponding to maximum height over the area covered by that pixel. If the current end of the ray is above the height, we can advance the ray to that the height.

To help comprehension, we will explain simple incorrect version first. We traverse the image pyramid from level 0 to level n. We sample a displacement value from the current pyramid map level at the current end of ray. If the sampled height is lower than the current ray's height, the ray advance to that height; otherwise we does not move the ray. In both case, we increase the level of the image pyramid. We repeat this work until the current level equals to a leaf level.

Sometimes, above explanation is incorrect. We said that if sampled height is lower than the current ray's height, the ray advance to the intersection position with sampled height. However, we cannot advance the ray over the boundary of pixel because we have no information out of current node. Thus if the ray is to run over the boundary of pixel, the ray must stop just over the boundary (see Figure 10). Pixel boundary position along the ray can be computed only addition [4].

There is a case that a ray cross both x and y plane at the same time. In this case, we simply select the nearest plane (see Figure 12).

```
level = ROOT_LEVEL;
d = tex2dlod(pyramid, float4(P.xyz, level));
P_prime = P + E * d;
level--;

while (level >= 0)
{
    d = tex2dlod(pyramid, float4(P.xyz, level));

    if (P_prime.z < d)
        P_prime = P + E * d;

    level--;
}
```

Figure 9. The pseudo-code for a ray/height-field intersection test using an image pyramid (simple version).

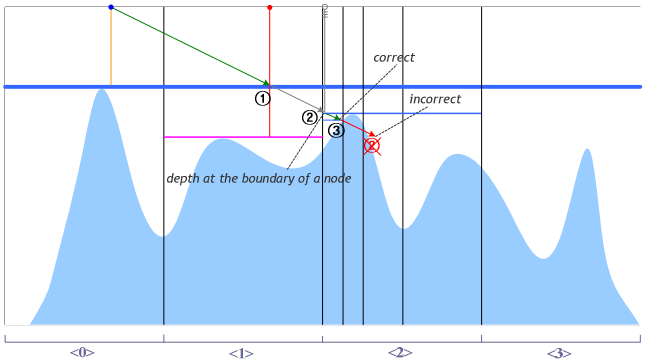

Figure 10. An example that ray runs over the pixel boundary. If we advance to position (2x)s, we will miss the intersection. We must advance to position (2) and restart the trace.

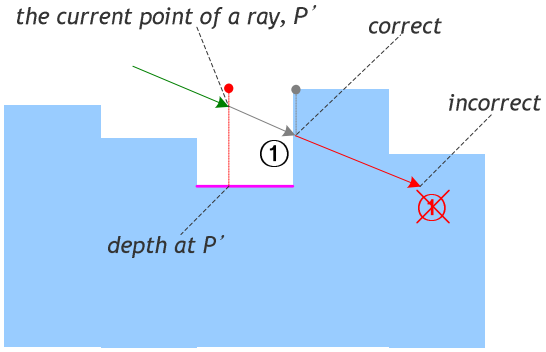

Figure 11. The example of intersection between a ray and the side of a height-field. By using node-crossing, the ray moves to the boundary of a node.

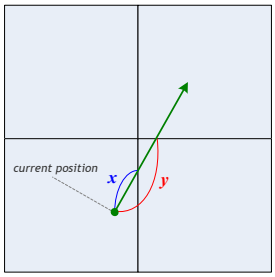

Figure 12. We select a plane that distance to a boundary is shorter than another (blue: x) if a ray crosses both boundaries of nodes through x and y planes.

```
level = ROOT_LEVEL;
d = tex2dlod(pyramid, float4(P.xyz, level));
P_prime = P + E * d;
level--;

while (level >= 0)
{
    d = tex2dlod(pyramid, float4(P.xyz, level));

    if (P_prime.z < d)
    {
        P_tmp = P + E * d;

        if (GetNodeNum(P_tmp) == GetNodeNum(P))
        {
            P_prime = P_tmp;
            level--;
        }
        else
        {
            P_prime = AcrossNode(P_prime) + offset;
        }
    }
    else
    {
        level--;
    }
}
```

Figure 13. A pseudo-code including node-crossing.

3.3 Bilinear Interpolation for Magnification

Until now, we described an issue only with nearest neighbor filetering. As a result it produces blocky images (see Figure 14's top). Unfortunately, we cannot find exact ray/height-field intersections just using bilinear interpolated pyramid map (see Figure 15).

We now introduce a new technique to solve this problem. At first, find the intersection point between the ray and point sampled height-field, *i.e,* perform the work described at sub-section 3.2, let us call this point P_1 and let's call the intersection point between the ray and bi-linearly interpolated height-field P_2.

If the ray intersects with the bi-linearly interpolated displacement map, the distance between P_1 and P_2 must be less than a half pixel in displacement map.

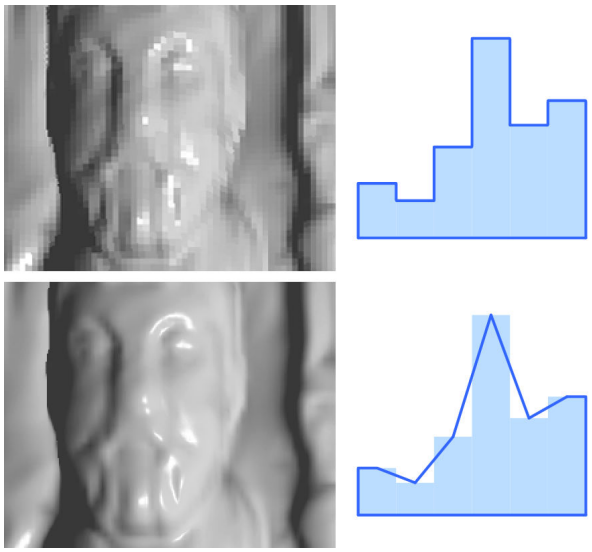

Figure 14. Comparison between rendering by nearest neighbor filtering (top) and bilinear filtering (bottom).

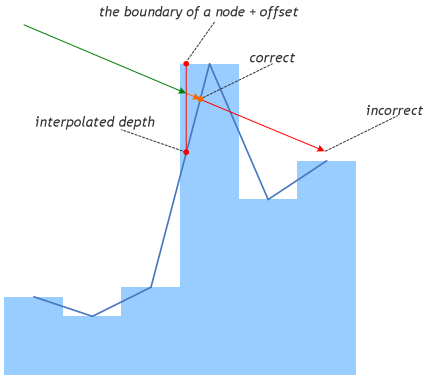

Figure 15. The cause of a visual artifact by using bilinear interpolation filtered a displacement map. If we use the interpolated map, a ray will advance to a red arrow.

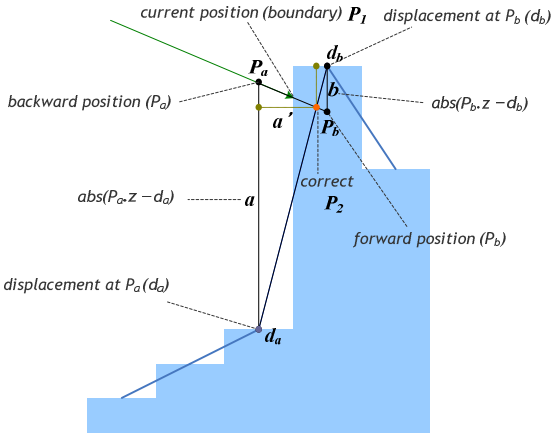

Figure 16. How to compute a correct intersection interpolated?

We sample the bilinearly filtered height values at two positions along the ray. One sample (P_a) is at backward position by one half pixel and one sample (P_b) is one half pixel forward position along the ray. P_2 can be computed using following equation. a and b is the difference between the sampled height and ray's height at each sample position.

$$P_2 = lerp(P_a, P_b, a/(a+b))$$

As shown in figure 17, all rays intersecting with point sampled height field does not intersect with bi-linearly interpolated height field. db is the depth value at Pb. If Pb's height is higher than db, there is no intersection between the ray and bi-linearly interpolated height field and then we retraverse the tree.

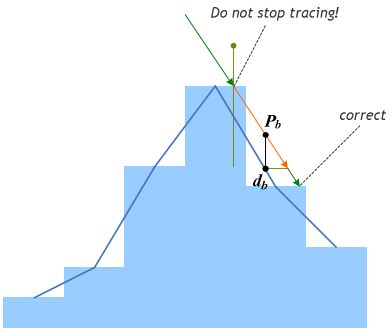

Figure 17. An example that ray intersecting with point sampled height-field does not intersect with bi-linearly interpolated height-field. At each leaf-level, we compute a forward position along the ray, P_b, and sample displacement d_b. If d_b is lower than $P_b.z$, we advance the ray according to d_b, and retraverse a tree at the current level.

Figure 18. Minification filtering by limit of max-traversal level (From left-top to right-bottom, we set limit of max-traversal level to 8, 7, 6, 5, 4 or 3). A surface goes flat by decreasing limit of max-traversal level.

3.4 Mip-map Filtering for Minification

In this subsection, we describe an issue of minification filtering to improve quality of images and to accelerate rendering speed when an object is far from the viewpoint. We perform top-down traversal of a quad-tree from the root to a leaf on an image pyramid; hence, our method fits to apply mip-mapping. In order to use mip-mapping, we simply limit max-traversal level of the tree (i.e., we stop traverse at a leaf as well as the max-traversal level). We can select a mip-level in a pixel shader like [17, 12] in real-time.

4. RESULTS

We have implemented pyramidal displacement mapping proposed in this paper using DirextX 9.0 with HLSL 3.0 on a ATI Radeon X1900 512MB graphics card. Other system configurations such as CPU do not affect performance because almost processes are executed on the GPU. Examples are rendered using a 256x256 texture except figure 4, 21, 22 and 23. We used textures with 512 by 512 texels for figure 4, and 1024 by 1024 resolution for figure 22 and 23. For figure 21, we used 256 by 256, 512 by 512, 1024 by 1024 and 2048 by 2048 resolution textures. Our method requires additional memory for the pyramid of displacement map.

Face images in figure 1 and 19 are rendered using a displacement map using real-time acquired depth and color images from [18].

We compared the result of our method using only 2 polygons to that of CPU based height mapping using 131,072 micro-polygonal meshes (figure 1's middle and figure 20). These figures show our method can present geometric details well. To render a 512 by 512 resolution image, our method needed only about 0.35MB memory, and rendered at 480.0fps (figure 20's left). Whereas, a CPU based height map rendering method (figure 20's right) were necessary about 2.1 MB memory for many polygons, and rendered at 50.13fps. As we use more detailed model, difference will be getting bigger.

Rendering performance of [13, 14] is susceptible to resolution of a displacement map. Figure 21 shows significance of our method against [13, 14]. Images are rendered at 480.06fps for a 256x256 displacement map, 432.29fps for a 512x512 map, 421.35fps for a 1024x1024 map, and 375.53fps for a 2048x2048 map respectively.

[10, 11] cause distracting artifacts on a sharp height-field or at steep grazing angles. Figure 23 shows our method can render artifacts-free images. Other solutions, [13, 14], worsen rendering performance as increasing resolution of a texture. Figure 22 and figure 23 are rendered using an easy-made displacement map. To make the map, we pasted a logo to an empty image and typed some text, and then converted to a grayscale image simply. It means that any person cans produce fascinating 3D images without any drawing (or modeling) techniques.

5. CONCLUSION

We introduced a novel GPU based method for rendering artifacts-free height-field in real-time, called pyramidal displacement mapping. We use a pyramidal data structure whose pixel value are minimum values over the area covered by it. The pyramid of displacement map forms a quad-tree. We traverse this quad-tree by top-down approach for finding ray/height-field intersections. Our method presented accurate surface details at steep grazing

angles even if a height-field is sharp or irregular. Moreover, the method was not sensitive to resolution of a displacement map. We also introduced techniques for magnification and minification filtering. They improved quality of synthetic images. Minification filtering also accelerated rendering speed. Pyramidal displacement mapping can present a complex geometry like micro-polygons using only few polygons. Developers can easily apply to their virtual reality applications, video games, etc.

Pyramidal displacement mapping is more accurate than other recently proposed techniques [10, 11, 12] that approximates intersection roughfly, but our method is slightly slower than ones. Our method can be excuted on graphics cards supports shader model 3.0 to fetch data from a LOD texture. This defect is obvious when we render an interpolated height-field (*i.e.*, magnification filtering). We would like to enhance ray/height-field intersection tests and magnification filtering. We will also develop a technique for rendering self-shadow, especially soft shadow like [12].

ACKNOWLEDGEMENT

We thank the authors of [18] for providing the color and depth face images in figures 1 and 19. This work was supported by the Korea Research Foundation Grant (KRF-2004-005-D00198).

REFERENCES

[1] Blinn, J. F. 1978. Simulation of Wrinkled Surfaces. *In Proceedings of the 5th annual conference on Computer graphics and interactive techniques*, ACM Press, pp. 286-292.

[2] Cook, R. L. 1984. Shade trees. *In Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, ACM Press, 223–231.

[3] Patterson, J.W., Hoggar. S. G., and Logie, J.R. 1991. Inverse Displacement Mapping. In EUROGRAPHICS Conference Proceedings. *Computer Graphics Forum* Volume 10, Issue 2 (1991) pp. 129-139.

[4] Cohen, D. and Shaked, A., Photo-Realistic Imaging of Digital Terrain, *Computer Graphics Forum*, 12, 3 (September 1993), 363-374.

[5] Wright, J. R., and Hsieh, J. C. L. A Voxel-Based, *Forward* Projection Algorithm for Rendering Surface and Volumetric Data. *Proc. IEEE Visualization `92*. IEEE Computer Society, Boston, MA, 1992, pp. 340-348.

[6] Lee, C.-H. and Shin, Y. G. A Terrain Rendering Method Using Vertical Ray Coherence. *Journal of Visualization and Computer Animation*, 8(2):97--114, 1997.

[7] Lee, C., and Shin, Y. G., An Efficient Ray Tracing Method for Terrain Rendering, *In proceedings of Pacific Graphics '95*, 1995, pp. 180-193.

[8] Musgrave, F. K. Grid tracing: Fast Ray Tracing for Height Fields. *Technical report*, Department of Mathematics, Yale University, December 1991.

[9] Coquillart, S., and Gangnet, M. 1984. Shaded display of digital maps. *IEEE Computer Graphics and Applications*, pages 35-42, July 1984.

[10] Brawley, Z., and Tatarchuk, N. 2004. Parallax Occlusion Mapping: Self-Shadowing, Perspective-Correct Bump Mapping Using Reverse Height Map Tracing. *In ShaderX3: Advanced Rendering with DirectX and OpenGL*, Engel, W., Ed., Charles River Media, pp. 135-154.

[11] Policarpo, F., Oliveira, M. M., and Comba, J. 2005. Real-Time Relief Mapping on Arbitrary Polygonal Surfaces. *In ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games Proceedings*, ACM Press, pp. 359-368.

[12] Natalya, M. 2006. Dynamic parallax occlusion mapping with approximate soft shadows. *In Proceedings of the 2006 symposium on Interactive 3D graphics and games*, ACM Press, pp.63-69.

[13] Donnelly, W. 2005. Per-Pixel Displacement Mapping with Distance Functions. *In GPU Gems 2*, M. Pharr, Ed., Addison-Wesley, pp. 123 – 136.

[14] Baboud, L., and Decoret, X. 2006. Rendering Geometry with Relief Textures. *In Graphics interface Conference Proceedings*.

[15] Samet. H. 1989. Applications of Spartial Data Structures. *Addison Wesley*, 1989.

[16] Kaneko, T., Takahei, T., Inami, M., Kawakami, N., Yanagida, Y., Maeda, T., and Tachi, S. 2001. Detailed Shape Representation with Parallax Mapping. *In Proceedings of the ICAT 2001*, 205--208.

[17] Shreinder, D., Woo, M., Neider, J., and Davis, T.. 2005. OpenGL® Programming Guide: The Official Guide to Learning OpenGL®, version 2, *Addison-Wesley*.

[18] Tsalakanidou, F., Forster, F., Malassiotis, S., and Strintzis, M. G. 2005. Real-time acquisition of depth and color images using structured light and its application to 3D face recognition. *Real-Time Imaging* 11, 5-6 (Oct. 2005), 358-369.

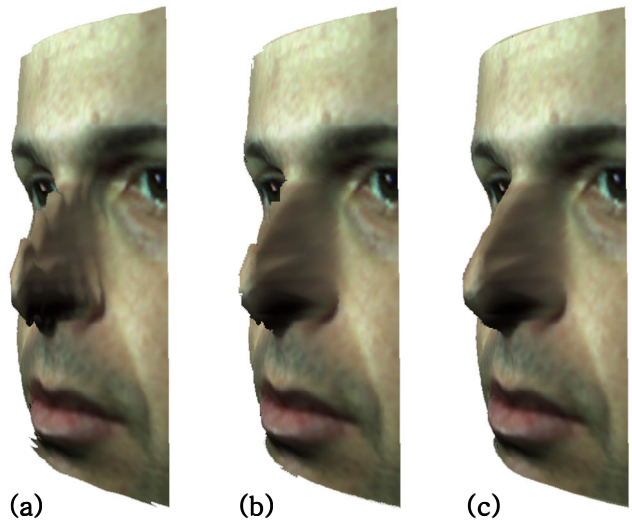

Figure 19. Parallax occlusion mapping (a) and relief mapping (b) cause serious artifacts. Our method is a great solution to produce different view images from a depth image (c).

Figure 20. Left: our method (480.06fps), right: micropolygons (50.13fps).

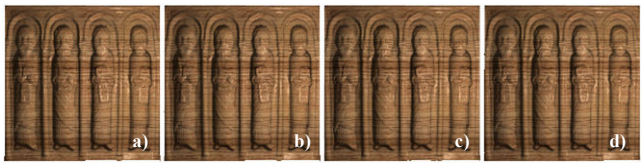

Figure 21. Results from variety of resolutions. A: 256x256 (480.06fps), b: 512x512 (437.29fps), c: 1024x1024 (421.35fps), d: 2048x2048 (375.53fps).

Figure 22. Pyramidal displacement mapping applied to an environment mapped transparent object.

Figure 23. The example of 3D reconstruction from a 2D logotype image. (top: parallax occlusion mapping, middle: relief mapping, bottom: pyramidal displacement mapping).