Accurate Per-Pixel Displacement Mapping using a Pyramid Structure

Hyunwoo Ki*
INNOACE Co., Ltd.

Kyoungsu Oh†
Soongsil University

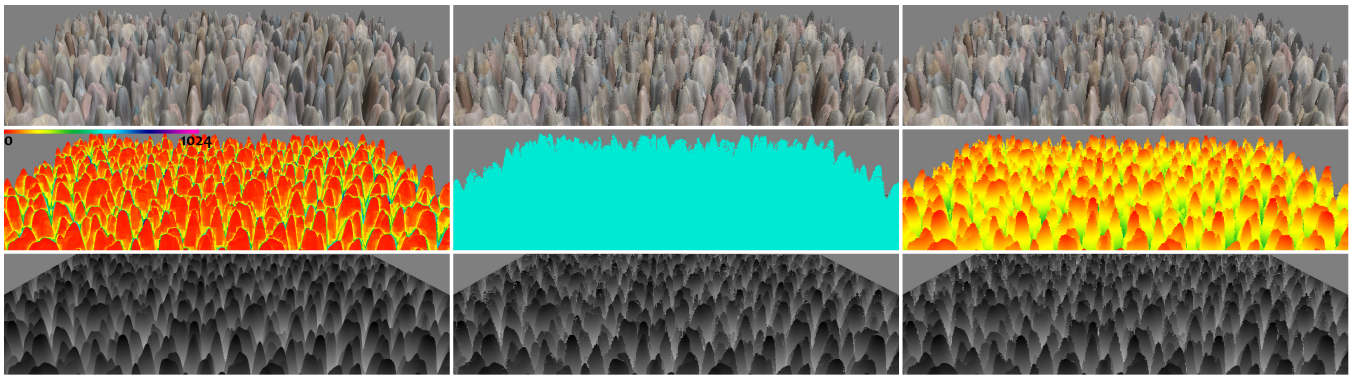

Figure 1: Various per-pixel displacement mapping are shown: our method (left), cone step mapping (center), relief mapping (right). Top images are results. Middle ones visualize the number of search steps. Bottom images show depth as the intersections.

Abstract

We present a per-pixel displacement mapping method that provides high accuracy as well as real-time performance. This arises from the observation that we can safely skip empty regions by moving a ray to the maximum height value. We store this bounding information in the image pyramid of a heighfield texture. In our method, a ray along view direction advances hierarchically via this pyramid structure. This method, called *pyramidal displacement mapping*, guarantees finding the correct intersections on any heightfield and viewing conditions. This article explains implementation details with more accessible illustrations and descriptions for 3D application developers. Additionally, we introduce a new LOD technique to improve rendering performance and quality.

Keywords: Per-pixel displacement mapping, image pyramid, LOD, real-time rendering, graphics hardware

1 Introduction

Per-pixel displacement mapping is a texture mapping method to represent complex surface details by displacing texture coordinate according to height profiles stored in a height-field texture (a.k.a displacement map, depth map, or relief map), without increasing geometric complexity. Today, many graphics people is interested in this method, because it can easily add very impressive details to the surfaces with low memory and cheap processing requirements. As the capabilities of graphics hardware are enhanced, many techniques have been newly presenting. In the per-pixel displacement mapping, we generally make a ray along view direction and then cast it to the heightfield in order to find ray-heightfield intersection points. It is the most important how we make the ray casting algorithm more "fast" and "accurate".

There are many related work about real-time ray casting on programmable graphics hardware: [Policarpo et al. 2005] uses a combination of linear and binary searches. [Tatarchuk 2006] refers the height profile to ray casting. However, these methods cannot guarantee finding the correct ray-heightfield intersection. Especially, aliasing artifacts are noticeable on spatially-varying height profiles, and grazing view angles. [Dummer 2006] compute the intersection using a precomputed cone map to prevent aliasing problem but this method causes a undesired distortion artifact. In order to reduce the artifacts of these methods, we should use more samples, but it drops rendering performance down. [Donnelly 2006; Baboud and Décoret 2006] store precomputed self-occlusion information in 3D and 2D textures, respectively. They require large data of a 3D texture or (and) lengthy precomputation times.

[Cohen and Shaked 1993] proposed a rapid hierarchical ray casting algorithm for terrain rendering on the CPU, using a image pyramid. The bounding information is stored in its mipmap chain. Recently, [Oh et al. 2006; Schroders and van Gulik 2006] implemented Cohen and Shaked [1993]'s algorithm on commodity graphics hardware. Leaf level pixels of the image pyramid store the original height values (i.e., same as the conventional heightfield texture). Each inner level pixel store maximum height (= minimum depth) of four corresponding pixels of child level. The root pixel (1×1) stores the top of the heightfield. Using this pyramid structure, we can safely skip empty regions and represent accurate self-occlusion and shadows on any conditions. Oh et al. [2006] also proposed a new linear filtering algorithm, which did not argued [Cohen and Shaked 1993; Schroders and van Gulik 2006]. This linear filtering algorithm improves visual quality. In this article, we describe the implementation details of Oh et al. [2006]'s pyramidal displacement mapping with more readable illustrations and descriptions than the original paper, and present a new level of detail algorithm to enhance rendering performance and quality. Also, this article shows Cg shader code segments for 3D application developers, students, etc. Visit Hyunwoo Ki's homepage (http://ki-h.com/) to get a shader for NVIDIA FX Composer 2.

*e-mail: kih@innoace.com; homepage: http://ki-h.com/
†e-mail: oks@ssu.ac.kr

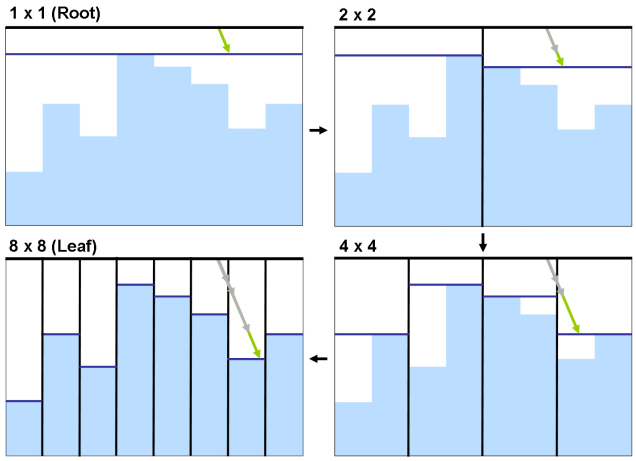

Figure 2: An overview of pyramidal displacement mapping. We cast a ray (arrow line) hierarchically in top-down order using a pyramid structure, which contains the children's maximum height value (the bound of a node; blue line).

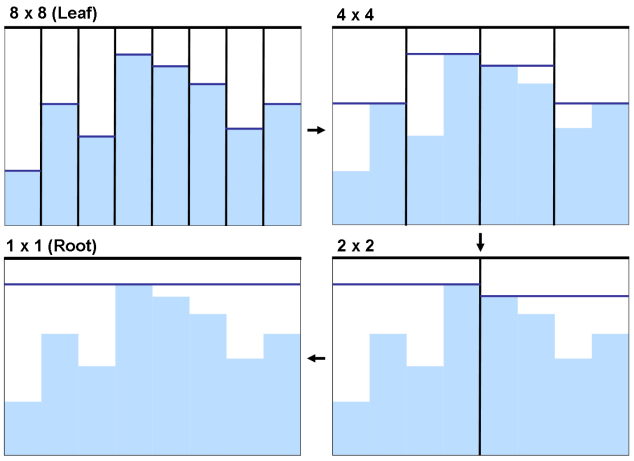

Figure 3: Building a pyramidal displacement map. With a bottom-up approach, we store the children's maximum height value of a heightfield texture in its mipmap chain.

2 Pyramidal Displacement Mapping

The algorithm lies on three steps: 1) building a pyramidal displacement map (PDM) as a preprocessing step, 2) tracing an eye ray using the PDM, and 3) computing illumination. This section describes implementation details with Cg code. Figure 2 illustrates our algorithm as an overview.

2.1 A Pyramid Structure

A PDM is a quadtree image pyramid. The image pyramid is a hierarchical collection of sublevel images from $2^0 \times 2^0$ to $2^n \times 2^n$, where n is the maximum level (i.e., the root). Each leaf pixel indicates a displacement value to actual surface. Each inner pixel, (i, j), stores the maximum height value of quadrant child pixels, $(2i, 2j)$, $(2i+1, 2j)$, $(2i, 2j+1)$, and $(2i+1, 2j+1)$, as illustrated in Figure 3. An inner pixel indicates the locally maximum height value. The root pixel denotes the globally maximum height value. We can perform this building step on the CPU as well as on the GPU. This step on the CPU spends approximately $1.8ms$ at 256×256 resolution, including copy between the CPU and the GPU (we tested on an NVIDIA GeForce 8800 GTX graphics card). We need to execute this step only when a displacement map is changed. Here is a pseudo-code for building a PDM.

```
// PDM building algorithm
for (int level = 1 to level <= max_level)
{
    for each pixel (i, j)
    {
        d0 = GetHeightAt(level - 1, 2i, 2j);
        d1 = GetHeightAt(level - 1, 2i + 1, 2j);
        d2 = GetHeightAt(level - 1, 2i, 2j + 1);
        d3 = GetHeightAt(level - 1, 2i + 1, 2j + 1);

        dest = min(min(min(d0, d1), d2), d3);
    }

    lvl++;
}
```

2.1.1 3D Texture Version

We can use a 3D texture instead of a 2D one as the PDM. A 2D PDM stores the maximum data in its mipmap chain, whereas a 3D PDM stores them in its depth (i.e., z coordinate or slice). We could use a $n \times n \times log(n)$ texture, but we prefer a $n \times n \times 2$ texture to reduce memory consumption. See Figure 4 as an example. Traditional pyramidal displacement mapping needs a function of LOD texture lookup (e.g., tex2Dlod function for Cg), provided in shader model 3.0 and 4.0. However, if we use the 3D texture, we can employ a lower version with static loop. Unfortunately, it spends more memory and is quite slow due to reading a 3D texture and adjusting texture coordinates.

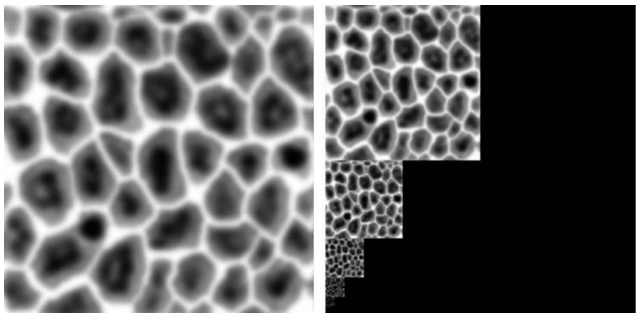

Figure 4: A PDM represented by a $n \times n \times 2$ texture.

2.2 Ray Casting within a Node

We will describe how to find intersection between a ray and displacement map using the PDM. We first set up the eye ray E that is a vector from the eye to a position transformed to tangent space. Tangent space is a space where x, y axis corresponds to texture coordinate and z coordinate is distance from the surface. And then scale this vector as $E = E/abs(E.z)$ (i.e., we make the $E.z$ to 1.0). Such scaling makes it easy to advance according to z value. More precisely, we can easily move the ray to a new position, P_2, corresponding to the displacement, d, as $P_2 = P + E * d$. Then, we trace the eye ray hierarchically with top-down order. We use nearest neighbor filtering of the PDM for the tracing to get the same displacement value per each texel as shown in Figure 5.

We start the ray casting with the input texture coordinate P interpolated through rasterization. At the root of the PDM, we read a displacement value d, and advance the ray to a new position corresponding to the d (see Figure 5.a). Next, at the advanced point P_2 and the next level, we read a new displacement, d, again. If d is bigger than $P_2.z$, it means that the ray is located above the bound of the node yet (i.e., not intersected), and thus we move the ray for-

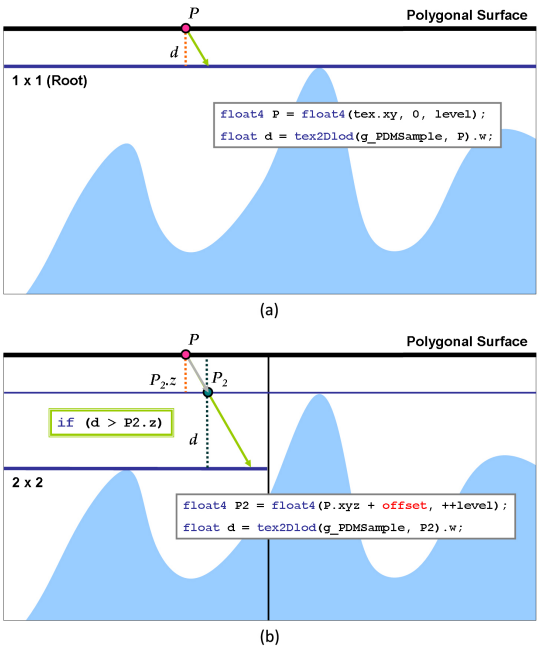

Figure 5: Ray casting within a node. We move the ray according to a height value fetched from the current level of the PDM, and then we repeat this process until the leaf level. The ray position at the leaf level will be the exact intersection.

ward again (see Figure 5.b). Otherwise, it means that the ray we reached the bound of the node, so we decrement the level by one without advancing the ray. We repeat this process until we reach leaf level. Here is a code sample. We wrote all code samples in this article in CgFX for NVIDIA FX Composer 2, and did not optimize the code to make it more readable.

```
// Ray casting within a node
float3 CastRayByPDM (float3 P,  // start point
                     float3 E)  // eye direction
{
    const int MAX_LEVEL = 8;    // mip-levels

    float3 P2 = P;              // current ray position
    int level = MAX_LEVEL;      // current level

    // Repeat to leaf level
    while (level >= 0)
    {
        float4 uv = float4(P2.xyz, level);
        float d = tex2Dlod(pyramid_map, uv).w;

        if (d > P2.z)
        {
            P2 = P + E * d;  // advancing
        }

        level--;    // leveling (to next level)
    }

    return P2;
}
```

2.3 Node Crossing

The height value is effective only in the current node. Therefore, if the ray advances crossing the boundary of a node, our tracing described the above section cannot guarantee finding the correct intersection. This occurs more frequently on spatially-varying height profiles and grazing view angles. To avoid missing the correct intersection, we check whether the advanced ray crosses the boundary

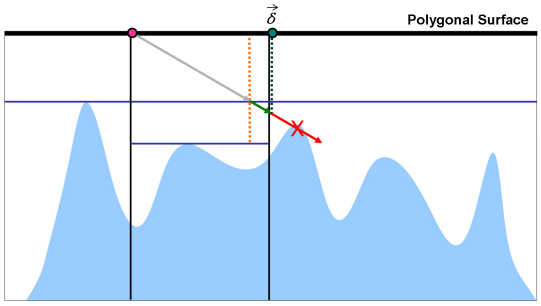

Figure 6: Node crossing. If a ray crosses the current node (red arrow), we move the ray to the boundary of the current node in the ray direction + δ (green arrow).

of the current node. If it crosses, we move the ray to a new position, $bn + \delta$. Here, bn is the boundary of the node at the level n in the ray direction, and delta is a tiny offset (e.g., half texel size scaled by a depth factor, at the leaf level). We call this step *node crossing*, and Figure 6 illustrates it. After the node crossing, we increment the level, and thus we trace the ray at on upper level.

There are two cases of a ray moved by the node crossing: first, the case that the displacement value at the moved ray position is smaller or equal to the ray's z value. In such case, the ray meets to the slope of the PDM, and thus the ray will not advance. Consequently, the current ray position will be the desired intersection accurately. The second case is that the displacement value at the moved ray position is bigger than the ray's z value. Therefore, the ray will advance according to displacements when we continue to trace the ray after the node crossing. Error caused by the node crossing equals to d, but the error does not be accumulated.

The node crossing can occur in both u and v coordinates. When a ray crosses the node boundary in both coordinates, we choose the nearest to the ray origin. We repeat such tracing including the node crossing until we arrive at the leaf level.

Here we explain how to check whether a ray crosses the current node's boundary or not. We know the current level's resolution. If we multiply a texture coordinate with a level's resolution and lower it to nearest integer (i.e., $floor$ operation), the texture coordinate is converted to an integer value. We can check whether the advanced position and current position are in the same node or not by comparing integer values.

```
float3 tmpP2 = P + E * d;   // new point (predictive)
float nodeCount = pow(2.0, float(MAX_LEVEL - level));
float4 nodeID = float4(P2.xy, tmpP2.xy);
nodeID = floor(nodeID * nodeCount);
float2 test = abs(nodeID.xy - nodeID.zw);
```

We can calculate the position of the boundary easily dividing the converted integer by the level's resolution. Here is a code sample of the node crossing.

```
if (test.x + test.y > 0.001)
{
    float texelSpan = 1.0 / nodeCount;

    float2 dirSign = (sign(E.xy) + 1.0f) * 0.5f;

    // distance to the next node's boundary
    float2 a = P2.xy - P.xy;
    float2 P3 = (nodeID.xy + dirSign) * texelSpan;
    float2 b = P3.xy - P.xy;

    // node crossing
    float2 dNC = (b.xy * P2.z) / a.xy;
    d = min(dNC.x, dNC.y) + fDeltaNC;

    level++;
}
```

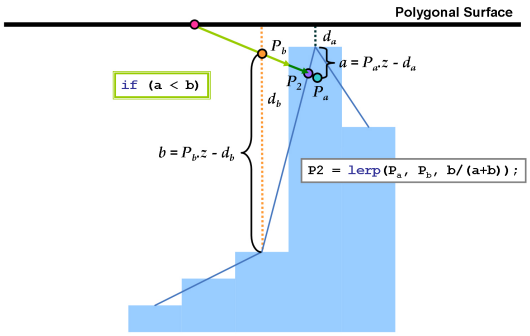

Figure 7: Linear filtering. After the search, we interpolate P_a and P_b linearly to represent smooth elevation.

2.4 Linear Filtering

Until now, we considered nearest neighbor filtering only for tracing. The result will be good, but we often prefer bilinear filtering to make elevation smooth. If the texel size of displacement map is bigger than pixel, stair-like shapes appear. Hence, we perform an additional refinement process at the leaf level to represent smooth elevation. We first sample an advanced position, P_a, along the ray as one half texel. At P_a, we read the interpolated height value, d_i, and then advance the ray according to the d_i. Then, we cast the ray with nearest filtering again, described in previous sections. We repeat such processing if the ray position is equal or below than the interpolated height value.

```
if (level <= 0)
{
    float2 ray2D =  P2.xy - P.xy;
    float rayLength = length(ray2D);

    float depthb = P2.z * (rayLength + TEXEL_SPAN_HALF);
    depthb /= rayLength;
    float4 P2b = float4(P + E * depthb, level);
    float depth2b = tex2Dlod(relief_map, P2b).w;

    if (depth2b > P2b.z)
    {
        P2 = P2b.xyz;
        level++;
    }
}
```

After this refinement step, we perform bilinear filtering manually as shown in Figure 7. We sample two positions along the ray by going forward and backward one half texel. The forward position is P_a, and the backward position is P_b. Then, we read the height d_a and d_b, and compute distance a and b. If a is smaller than b, we interpolate between P_a and P_b linearly using a code segment as "lerp(Pa, Pb, b/(a+b))", to estimate the new ray position, P_2, interpolated.

```
float2 ray2D = P2.xy - P.xy;
float rayLength = length(ray2D);

float deptha = P2.z * (rayLength - TEXEL_SPAN_HALF);
float depthb = P2.z * (rayLength + TEXEL_SPAN_HALF);
deptha /= rayLength; depthb /=
rayLength;

float4 P2a = float4(P + E * deptha, 0);
float4 P2b = float4(P + E * depthb, 0);
float depth2a = tex2Dlod(pyramid_linear_map, P2a).w;
float depth2b = tex2Dlod(pyramid_linear_map, P2b).w;

float la = abs(P2a.z - depth2a);
float lb = abs(P2b.z - depth2b);
P2 = lerp(P2a.xyz, P2b.xyz, la / (la + lb));
```

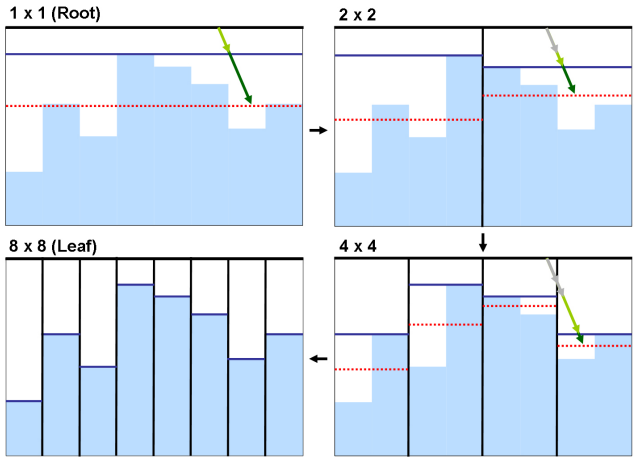

Figure 8: Level of detail algorithm. We perform the conventional ray casting algorithm with the maximum height values until the selected mip-level (pea-green arrow), then perform ray casting with the mipmaped height values instead of the maximum ones (green arrow). Blue solid and red dotted lines indicate the maximum and mippaed height, respectively.

Our solution is good approximation assuming that a displaced surface smaller than a texel is linear along the ray. In order to get an exact solution, we would have to calculate intersection between a ray and bilinear patch using an analytic manner such as [Ramsey et al. 2004].

2.5 Level of Detail

Level of detail (LOD) is an important technique that involves decreasing the complexity of an object as it moves far away from the view-point (or sometimes according object importance), to improve rendering performance with negligible loss of visual quality. Ray casting algorithm is fast and efficient but performing this algorithm for every pixel wastes cost. [Tatarchuk 2006] employed a simple approach that uses her parallax occlusion mapping for close surfaces and conventional normal mapping for distant surfaces. Highly related, previous work such as [Cohen and Shaked 1993; Schroders and van Gulik 2006] did not describe a LOD algorithm. The original pyramidal displacement mapping [Oh et al. 2006] stopped ray casting at the specified mip-level. However, it is not the low-frequency representaion of the heightfield and thus exhibits poor visual quality. This subsection presents a new LOD algorithm for the pyramidal displacement mapping.

The algorithm is based on the observation that the mipmaped value of certain height values is always less than or equal to the maximum height value stored in the PDM, at the same level. Therefore, we first do our ray casting algorithm according to the maximum height value, until the selected mip-level as described above subsections. Then, we do the ray casting again but we consider the mipmaped values instead of the maximum ones. This technique also guarantees correct intersections, but will suffer from aliasing. There are several code changes.

```
// 1) Loop
while (level >= 0)
-> while (level >= lodlevel)

// 2) Fetch
float d = tex2Dlod(pyramid_map, uv).w;
-> float d = (level <= lod) ? tex2Dlod(pyramid_lodmap, uv).w: \
tex2Dlod(pyramid_map, uv).w;

// 3) Refinement: Insert this code before leveling
```

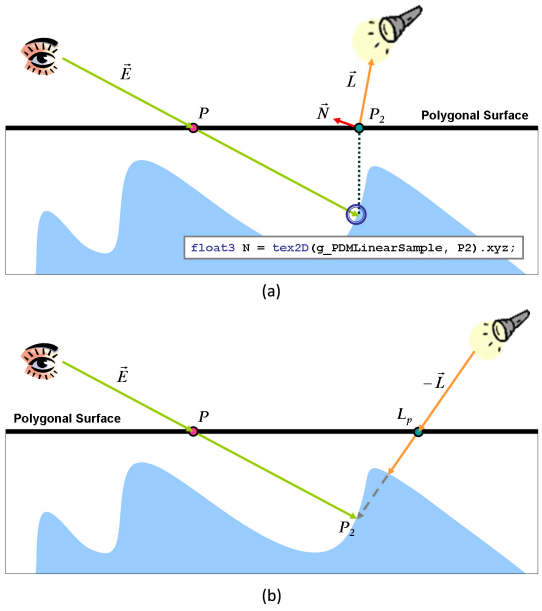

Figure 9: Shading and self-shadowing. We compute illumination using the normal at P_2 instead of P (a). In order to represent self-shadow, we cast the light ray through the PDM and then compare the first-intersection with P_2 (b). If the intersection from the light differs from P_2, we consider that the pixel being shaded as in shadowed.

```
if (level <= lod)
{
    float d = tex2Dlod(pyramid_map, uv).x;

    if (d > P2.z)
        level++;
}
```

2.6 Shading and Self-Shadowing

After finding intersection, we compute illumination using the normal at P_2 (see Figure 9.a). In order to represent self-shadow, we search intersection between the heightfield and a ray from the light to the current intersection point like [Policarpo et al. 2005]. First, we calculate intersection point L_p between plane $z = 0$ and the ray from P_2 to the light source. Then we cast the ray from L_p to P_2 though the PDM to find the intersection. If two intersections from the eye and light rays are different to each other, we consider that a pixel is in shadow (see Figure 9.b). Since our ray casting algorithm is more robust than [Policarpo et al. 2005]'s one, we can also represent exact self-shadow in any view and lighting conditions.

3 Results

We tested our and related methods on an NVIDIA GeForce 8800 GTX graphics card. We used target profiles vp40 and fp40, and a pregenerated PDM as a texture unit for FX Composer 2. Figure 1, 10 and 11 show side-by-side comparison among various techniques. Texture mapping adds only color details that are independent of lighting and viewing conditions, to the surface. Normal mapping known as bump mapping adds the appearance of geometric details by changing the surface normals, which affect the lighting result, but it makes the surfaces flat at grazing view angles. Parallax occlusion mapping and relief mapping provides impressive elevation with self-shadow and motion parallax that is the apparent shift of an object caused by changing view position or angle. However, these methods cause an aliasing artifact on spatially-varing

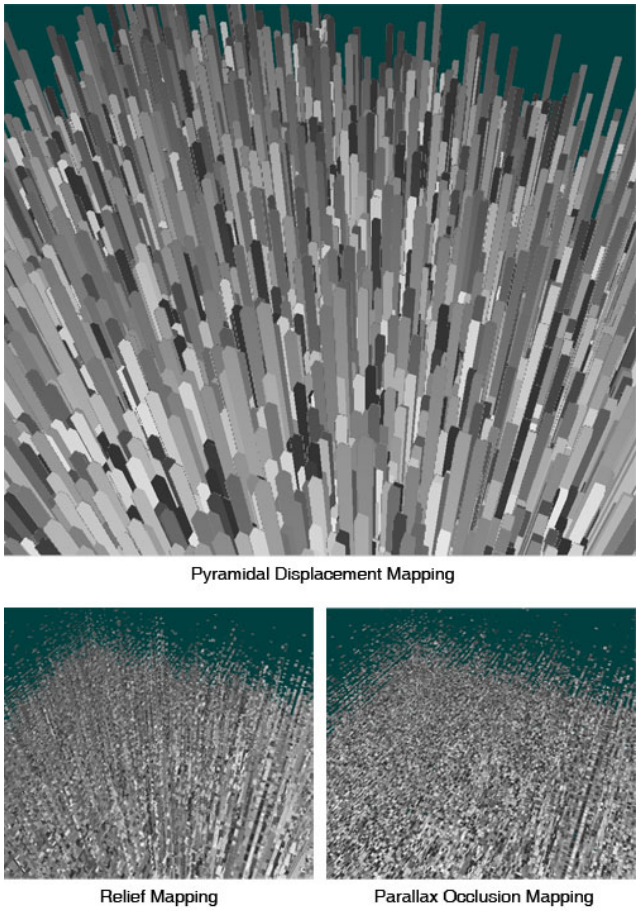

Figure 10: Pyramidal displacement mapping can correctly find intersections on a very spatially-varying heightfield texture.

heightfields or at grazing view angles due to their non-robust linear search. Cone step mapping prevents the aliasing artifact but causes a distortion artifact when we use small numbers of search steps. Whereas pyramidal displacement mapping represents accurate elevation and self-shadow without such artifacts. For Figure 11, we use 0.35 of depth scaling (i.e., uniform float d_Factor), 100 of linear steps and 7 of binary steps for relief mapping, and 100 of cone steps for cone step mapping. The pixel color indicates the height found (i.e., $P2.z$). Table 1 displays the time performance of our pyramidal displacement mapping with varying 256×256 heightfield textures on 512×512 screen resolution at grazing view angles. In the best case, our ray casting iterates $log(n)$ times, where n is the resolution of a heightfield texture. At such grazing angles, it would, however, perform the additional, costly node crossing and refinement steps for linear interpolation many times. Nevertheless, we still archived high-frame rates 400 to 500 fps for nearest neighbor filtering and 200 to 300 fps for linear filtering.

Figure 13 shows a Fresnel effect including environment lighting for a texted logo. Figure 13 exhibits an effect of our LOD algorithm described in subsection 2.5 (linear filtering is not applied). This texture has 512×512 texels. Far surface regions are rendered with a low-detailed heightfield relatively. This LOD algorithm improved rendering performance approximately 2 times. Figure 14 shows results with varying a LOD level. This texture has 256×256 texels.

Table 1: Performance - Filtering Types (fps)

Heightfield	Nearest	Linear
Hawaii	482.27	285.88
Rockbump	494.89	292.84
NVIDIA logo	521.56	306.80

4 Conclusion

We have presented a method that enables fast and correct rendering of displaced surfaces in the pixel level. We use a pyramid structure to skip empty regions safely and efficiently. Our method can be easily implemented and we achieved real-time frame rates. We showed a simple LOD algorithm for pyramidal displacement mapping, but it is hard to do trilinear filtering. The presented method should need functions of shader model 3.0 or higher for dynamic branching, and LOD texture fetch to select a mip-level arbitrarily. This method can combine with [Oliveira and Policarpo 2005] to represent silhouette. The implementation of the technique is provided on the web. This code is based on a Policarpo [2005]'s relief mapping sample. You can interactively change a shader (normal mapping, parallax occlusion mapping, cone step mapping, relief mapping, or pyramidal displacement mapping), textures, view and lighting conditions, etc.

References

BABOUD, L., AND DÉCORET, X. 2006. Rendering geometry with relief textures. In *Graphics Interface '06*.

COHEN, D., AND SHAKED, A. 1993. Photo-realistic imaging of digital terrains. *Computer Graphics Forum 12*, 3 (August), 363–373.

DONNELLY, W. 2006. *Per-pixel displacement mapping with distance functions*. GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation. NVIDIA, ch. 13, 123–146.

DUMMER, J. 2006. Cone step mapping: An iterative rayheightfield intersection algorithm. http://www.lonesock.net/files/ConeStepMapping.pdf.

OH, K., KI, H., AND LEE, C.-H. 2006. Pyramidal displacement mapping: a gpu based artifacts-free ray tracing through an image pyramid. In *VRST '06: Proceedings of the ACM symposium on Virtual reality software and technology*, 75–82.

OLIVEIRA, M. M., AND POLICARPO, F. 2005. An efficient representation for surface details. Tech. rep., UFRGS.

POLICARPO, F., OLIVEIRA, M. M., AND COMBA, J. 2005. Real-time relief mapping on arbitrary polygonal surfaces. In *I3D '05: Proceedings of the 2005 symposium on Interactive 3D graphics and games*, 155–162.

RAMSEY, S. D., POTTER, K., AND HANSEN, C. 2004. Ray bilinear patch intersections. *journal of graphics tools 9*, 3, 41–47.

SCHRODERS, M. F. A., AND VAN GULIK, R. 2006. Quadtree relief mapping. In *Graphics Hardware 2006*, 61–66.

TATARCHUK, N. 2006. Dynamic parallax occlusion mapping with approximate soft shadows. In *I3D '06: Proceedings of the 2006 symposium on Interactive 3D graphics and games*, 63–69.

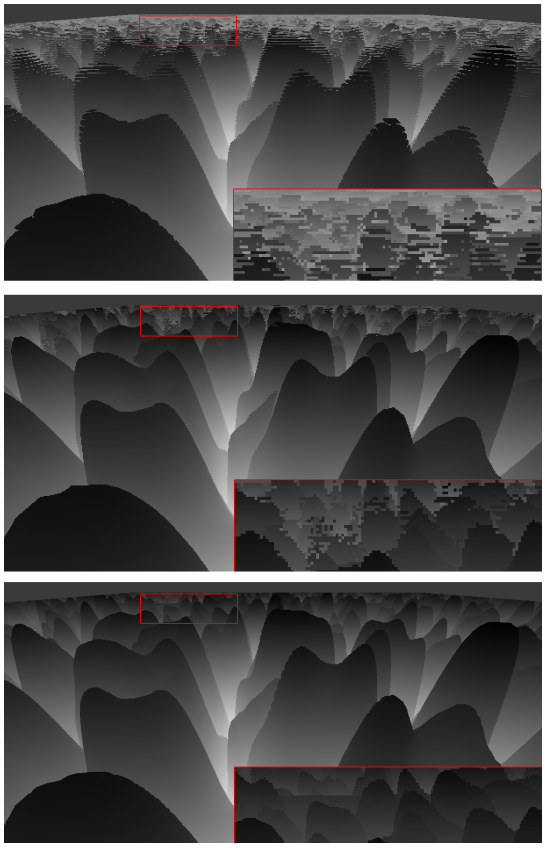

Figure 11: Qualitative comparison: relief mapping with 100 linear and 7 binary steps (top), cone step mapping with 100 cone steps (middle), and pyramidal displacement mapping (bottom). The pixel color indicates the height of the intersection, $P_2.z$.

Figure 12: NVIDIA logo with refraction and reflection effects.

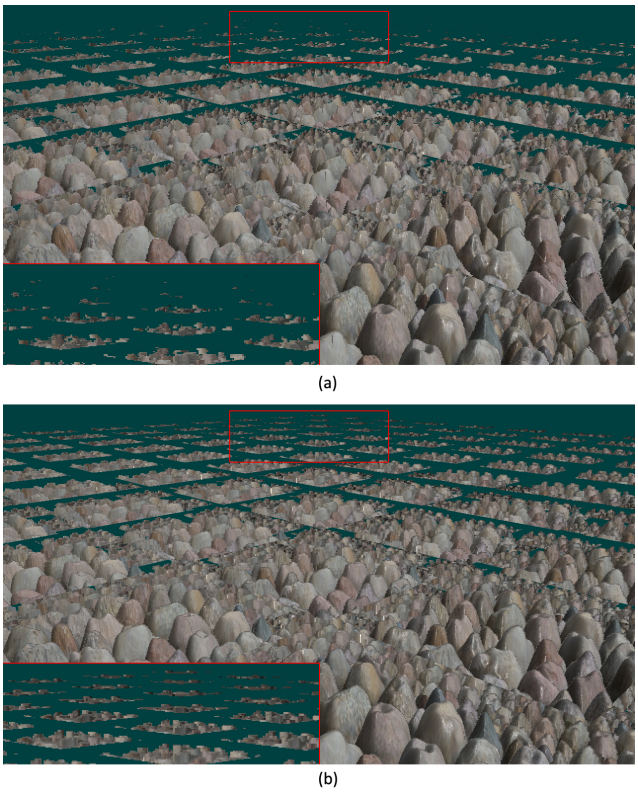

Figure 13: Conventional relief mapping (a) and our pyramidal displacement mapping with level of detail (b), at same frame rates. Linear filtering is not applied.

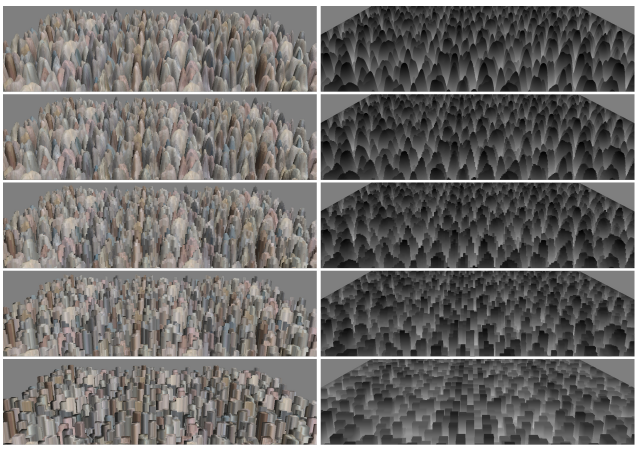

Figure 14: LOD results (0 to 4).