Chapter 5

Practical Parallax Occlusion Mapping with Approximate Soft Shadows for Detailed Surface Rendering

Natalya Tatarchuk[7]
ATI Research

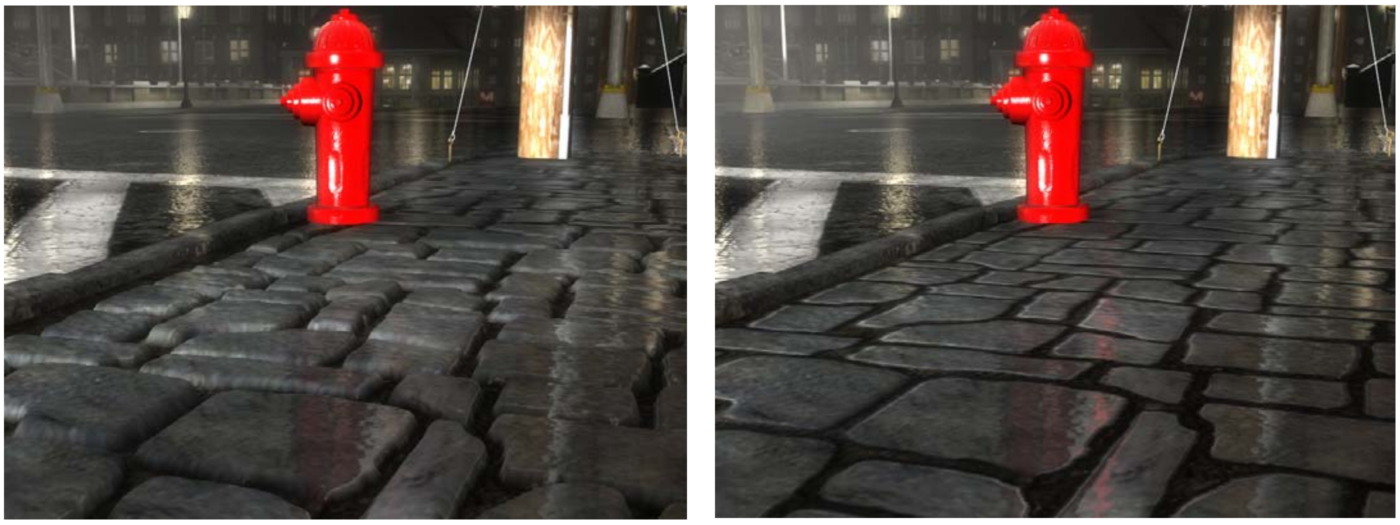

(a) (b)

***Figure 1**. Realistic city scene rendered using parallax occlusion mapping applied to the cobblestone sidewalk in (a) and using the normal mapping technique in (b).*

[7] natasha@ati.com

5.1 Abstract

This chapter presents a per-pixel ray tracing algorithm with dynamic lighting of surfaces in real-time on the GPU. First, we will describe a method for increased precision of the critical ray-height field intersection and adaptive height field sampling. We achieve higher quality results than the existing inverse displacement mapping algorithms. Second, soft shadows are computed by estimating light visibility for the displaced surfaces. Third, we describe an adaptive level-of-detail system which uses the information supplied by the graphics hardware during rendering to automatically manage shader complexity. This LOD scheme maintains smooth transitions between the full displacement computation and a simplified representation at a lower level of detail without visual artifacts. Finally, algorithm limitations will be discussed along with the practical considerations for integration into game pipelines. Specific attention will be given to the art asset authoring, providing guidelines, tips and concerns. The algorithm performs well for animated objects and supports dynamic rendering of height fields for a variety of interesting displacement effects. The presented method is scalable for a range of consumer grade GPU products. It exhibits a low memory footprint and can be easily integrated into existing art pipelines for games and effects rendering.

5.2 Introduction

The advances in the programmability of commodity GPUs in the recent years have revolutionized the visual complexity of interactive worlds found in games or similar real-time applications. However, the balance between the concept and realism dictates that in order to make the objects in these virtual worlds appear photorealistic, the visual intricacy demands a significant amount of detail. Simply painting a few broken bricks will not serve the purpose of displaying a dilapidated brick wall in a forlorn city any longer. With the raised visual fidelity of the latest games, the player wants to be immersed in these worlds – they want to *experience* the details of their environments. That demands that each object maintains its three-dimensional appearance accurately regardless of the viewing distance or angle.

Which brings us to an age-old problem of computer graphics - how do we render detailed objects with complex surface detail without paying the price on performance? We must balance the desire to render intricate surfaces with the cost of the millions of triangles associated with high polygonal surfaces typically necessary to represent that geometry. Despite the fact that the geometric throughput of the graphics hardware has increased immensely in recent years, there still exist many obstacles in throwing giant amounts of geometry onto the GPU. There is an associated memory footprint for storing large meshes (typically measured in many megabytes of vertex and connectivity data), and the performance cost for vertex transformations and animations of those meshes.

If we want the players to think they're near a brick wall, it should look and behave like one. The bricks should have deep grooves, an assortment of bumps and scratches. There should be shadows between individual bricks. As the player moves around the object, it needs to maintain its depth and volume. We wish to render these complex surfaces, such as this mythical brick wall, accurately – which means that we must do the following:

- Preserve depth at all angles
- Support dynamic lighting
- Self occlusions on the surface must result in correct self-shadowing on the surface without aliasing.

Throughout history, artists specialized in creating the illusion of detail and depth without actually building a concrete model of reality on the canvas. Similarly, in computer graphics we frequently want to create a compelling impression of a realistic scene without the full cost of complex geometry. Texture mapping is essential for that purpose - it allows generation of detail-rich scenes without the full geometric content representation. Bump mapping was introduced in the early days of computer graphics in [Blinn 1978] to avoid rendering high polygonal count models.

Bump mapping is a technique for making surfaces appear detailed and uneven by perturbing the surface normal using a texture. This approach creates a visual illusion of the surface detail that would otherwise consume most of a project's polygon budget (such as fissures and cracks in terrain and rocks, textured bark on trees, clothes, wrinkles, etc). Since the early days, there have been many extensions to the basic bump mapping technique including emboss bump mapping, environment map bump mapping, and the highly popular dot product bump mapping (normal mapping). See [Akenine-Möller02] for a more detailed description of these techniques. In Figure 1b above, we can see the per-pixel bump mapping technique (also called *normal mapping*) applied to the cobblestone sidewalk.

Despite its low computational cost and ease of use, bump mapping fails to account for important visual cues such as shading due to interpenetrations and self-occlusion, nor does it display perspective-correct depth at all angles. Since the bump mapping technique doesn't take into consideration the geometric depth of the surface, it does not exhibit parallax. This technique displays various visual artifacts, and thus several approaches have been introduced to simulate parallax on bump mapped geometry. However, many of the existing parallax generation techniques cannot account for self-occluding geometry or add shadowing effects. Indeed, shadows provide a very important visual cue for surface detail.

The main contribution of this chapter is an advanced technique for simulating the illusion of depth on uneven surfaces without increasing the geometric complexity of rendered objects. This is accomplished by computing a perspective-correct representation maintaining accurate parallax by using an inverse displacement mapping technique. We also describe a method for computing self-shadowing effects for self-occluding objects. The resulting approach allows us to simulate pseudo geometry displacement in the pixel shader instead of modeling geometric details in the polygonal mesh. This allows us to render surface detail providing a convincing visual impression of depth from varying viewpoints, utilizing the programmable pixel pipelines of commercial graphics hardware. The results of applying parallax occlusion mapping can be seen in Figure 1a above, where the method is used to render the cobblestone sidewalk.

We perform per-pixel ray tracing for inverse displacement mapping with an easy-to-implement, efficient algorithm. Our method allows interactive rendering of displaced surfaces with dynamic lighting, soft shadows, self-occlusions and motion parallax. Previous methods displayed strong aliasing at grazing angles, thus limiting potential applications' view angles, making these approaches impractical in realistic game scenarios. We present a significant improvement in the rendering quality necessary for production level results. This work has been originally presented in [Tatarchuk06].

Our method's contributions include:

- A high precision computation algorithm for critical height field-ray intersection and an adaptive height field sampling scheme, well-designed for a range of consumer GPUs (Section 6.5.1). This method significantly reduces visual artifacts at oblique angles.
- Estimation of light visibility for displaced surfaces allowing real-time computation of soft shadows due to self-occlusion (Section 6.5.2).
- Adaptive level-of-detail control system with smooth transitions (Section 6.5.3) for controlling shader complexity using per-pixel level-of-detail information.

The contributions presented in this chapter are desired for easy integration of inverse displacement mapping into interactive applications such as games. They improve resulting visual quality while taking full advantage of programmable GPU pixel and texture pipelines' efficiency. Our technique can be applied to animated objects and fits well within established art pipelines of games and effects rendering. The algorithm allows scalability for a range of existing GPU products.

5.3 Why reinvent the wheel? Common artifacts and related work

Although standard bump mapping offers a relatively inexpensive way to add surface detail, there are several downsides to this technique. Common bump mapping approaches lack the ability to represent view-dependent unevenness of detailed surfaces, and therefore fail to represent motion parallax—the apparent displacement of the object due to viewpoint change. In recent years, new approaches for simulating displacement on surfaces have been introduced. [Kaneko01] and [Welsh03] describe an approach for parallax mapping for representing surface detail using normal maps, while [Wang03] introduced a technique for view-dependent displacement mapping which improved on displaying surface detail as well as silhouette detail.

Displacement mapping, introduced by [Cook84], addressed the issues above by actually modifying the underlying surface geometry. Ray-tracing based approaches dominated in the offline domain [Pharr and Hanrahan 1996; Heidrich and Seidel 1998]. These methods adapt poorly to current programmable GPUs and are not applicable to the interactive domain due to high computational costs. Additionally, displacement mapping requires fairly highly tessellated models in order achieve satisfactory results, negating the polygon-saving effect of bump mapping.

Other approaches included software-based image-warping techniques for rendering perspective-correct geometry [Oliveira et al. 2000] and precomputed visibility information [Wang et al. 2003; Wang et al. 2004; Donnelly 2005]. [Wang03] describes a per-pixel technique for self-shadowing view-dependent rendering capable of handling occlusions and correct display of silhouette detail. The precomputed surface description is stored in multiple texture maps (the data is precomputed from a supplied height map). The view-dependent displacement mapping textures approach displays convincing parallax effect by storing the texel relationship from several viewing directions. However, the cost of storing multiple additional texture maps for surface description is prohibitive for most real-time applications. Our proposed method requires a low memory footprint and can be used for dynamically rendered height fields.

Recent inverse displacement mapping approaches take advantage of the parallel nature of GPU pixel pipelines to render displacement directly on the GPU ([Doggett and Hirche 2000; Kautz and Seidel 2001; Hirche et al. 2004; Brawley and Tatarchuk 2004; Policarpo et al. 2005]. One of the significant disadvantages of these approaches is the lack of correct object silhouettes since these techniques do not modify the actual geometry. Accurate silhouettes can be generated by using view-dependent displacement data as in [Wang et al. 2003; Wang et al. 2004] or by encoding the surface curvature information with quadric surfaces as in [Oliveira and Policarpo 2005].

Another limitation of bump mapping techniques is the inability to properly model self-shadowing of the bump mapped surface, adding an unrealistic effect to the final look. The horizon mapping technique ([Max 1988], [Sloan and Cohen 2000]) allows shadowing bump mapped surfaces using precomputed visibility maps. With this approach, the height of the shadowing horizon at each point on the bump map for eight cardinal directions is encoded in a series of textures which are used to determine the amount of self-shadowing for a given light position during rendering. A variety of other techniques were introduced for this purpose, again, the reader may refer to an excellent survey in [Akenine-Möller02].

A precomputed three-dimensional distance map for a rendered object can be used for surface extrusion along a given view direction ([Donnelly 2005]). This technique stores a 'slab' of distances to the height field in a volumetric texture. It then uses this distance field texture to perform ray "walks" along the view ray to arrive at the displaced point on the extruded surface. The highly prohibitive cost of a 3D texture and dependent texture fetches' latency make this algorithm less attractive and in many cases simply not applicable in most real-time applications. Additionally, as this approach does not compute an accurate intersection of the rays with the height field, it suffers from aliasing artifacts at a large range of viewing angles. Since the algorithm requires precomputed distance fields for each given height field, it is not amenable for dynamic height field rendering approaches.

Mapping relief data in tangent space for per-pixel displacement mapping in real-time was proposed in [Brawley and Tatarchuk 2004; Policarpo et al. 2005; McGuire and McGuire 2005] and further extended in [Oliveira and Policarpo et al. 2005] to support silhouette generation. The latter work was further extended to support rendering with non-height field data in [Policarpo06]. These methods take advantage of the programmable pixel pipeline efficiency by performing height field-ray intersection in the pixel shader to compute the displacement information. These approaches generate dynamic lighting with self-occlusion, shadows and motion parallax. Using the visibility horizon to compute hard shadows as in [Policarpo et al. 2005; McGuire and McGuire 2005; Oliveira and Policarpo

2005] can result in shadow aliasing artifacts. All of the above approaches exhibit strong aliasing and excessive flattening at steep viewing angles. No explicit level of detail schemes were provided with these approaches, relying on the texture filtering capabilities of the GPUs.

Adaptive level-of-detail control systems are beneficial any computationally intensive algorithm and there have been many contributors in the field of rendering. A level of detail system for bump mapped surfaces using pre-filtered maps was presented in [Fournier 92]. RenderMan® displacement maps were automatically converted to bump maps and BRDF representations in [Becker and Max 1993]. An automatic shader simplification system presented in [Olano et al. 2003] uses controllable parameters to manage system complexity. The resulting level-of-detail shader appearance is adjustable based on distance, size, and importance and given hardware limits.

5.4 Parallax Occlusion Mapping

This section will provide a brief overview of concepts of the parallax occlusion mapping method. We encode the displacement information for the surface in a height map as shown in Figure 2b. The inherent planarity of the tangent space allows us to compute displacement for arbitrary polygonal surfaces. Height field-ray intersections are performed in tangent space. The lighting can be computed in any space using a variety of illumination models. Efficient GPU implementation allows us to compute per-pixel shading, self-occlusion effects, and soft shadows, dynamically scaling the computations.

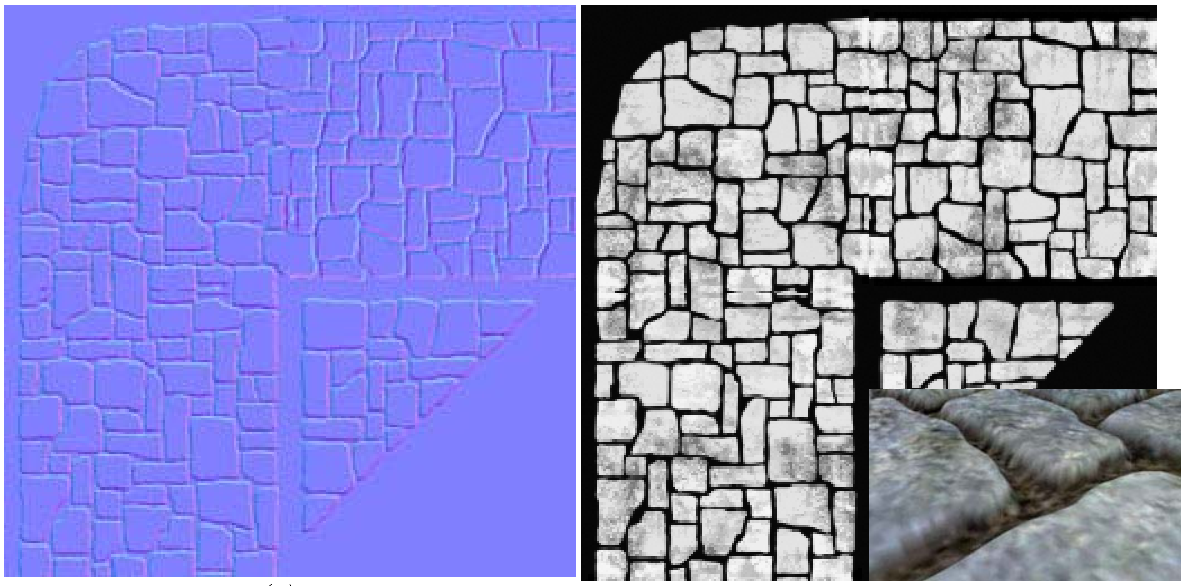

(a) *(b)*

Figure 2. *(a) Tangent space normal map used to render the cobblestone sidewalk in Figure 1a. (b) Corresponding height field encoding the displacement information in the range [0;1] for that sidewalk object and a close-up view of the rendered sidewalk*

The effect of motion parallax for a surface can be computed by applying a height map and offsetting each pixel in the height map using the geometric normal and the view vector. As we move the geometry away from its original position using that ray, the parallax is obtained by the fact that the highest points on the height map would move the farthest along that ray and the lower extremes would not appear to be moving at all. To obtain satisfactory results for true perspective simulation, one would need to displace every pixel in the height map using the view ray and the geometric normal. We trace a ray through the height field to find the closest visible point on the surface.

The input mesh provides the reference plane for displacing the surface downwards. The height field is normalized for correct ray-height field intersection computation (0 representing the reference polygon surface values and 1 representing the extrusion valleys).

The parallax occlusion mapping algorithm execution can be summarized as follows:

- Compute the tangent-space viewing direction $\hat{V}_{ts}$ and the light direction $\hat{L}_{ts}$ per-vertex, interpolate and normalize in the pixel shader
- Compute the parallax offset vector P (either per-vertex or per-pixel) to determine the maximum visual offset in texture-space for the current level (as described in [Brawley and Tatarchuk 2004])

In the pixel shader:

- Ray cast the view ray $\hat{V}_{ts}$ along P to compute the height profile–ray intersection point. We sample the height field profile along P to determine the correct displaced point on the extruded surface. This yields the texture coordinate offset necessary to arrive at the desired point on the extruded surface as shown in Figure 3. We add this parallax offset amount to the original sample coordinates to yield the shifted texture coordinates t_{off}

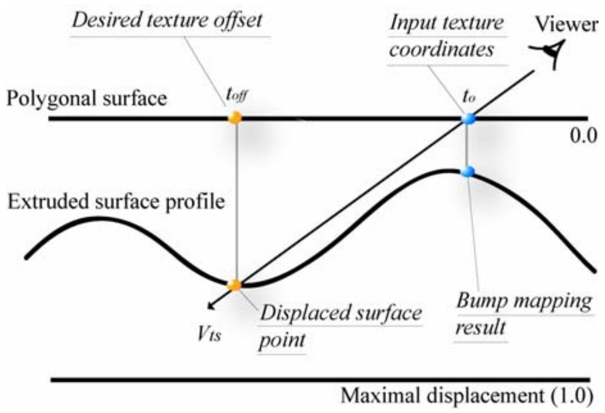


Figure 3. *Displacement based on sampled height field and current view direction.*

- Estimate the light visibility coefficient v by casting the light direction ray $\hat{L}_{ts}$ and sampling the height profile for occlusions.
- Shade the pixel using v, $\hat{L}_{ts}$ and the pixel's attributes (such as the albedo, color map, normal, etc.) sampled at the texture coordinate offset t_{off}.

Figure 6 later in the chapter illustrates the process above for a given pixel on a polygonal face. We will now discuss each of the above steps in greater detail.

5.4.1 Height Field – Ray Intersection

Techniques such as [Policarpo et al. 2005; Oliveira and Policarpo 2005] determine the intersection point by a combination of linear and binary search routines. These approaches sample the height field as a piecewise constant function. The linear search allows arriving at a point below the extruded surface intersection with the view ray. The following binary search helps finding an approximate height field intersection utilizing bilinear texture filtering to interpolate the intersection point.

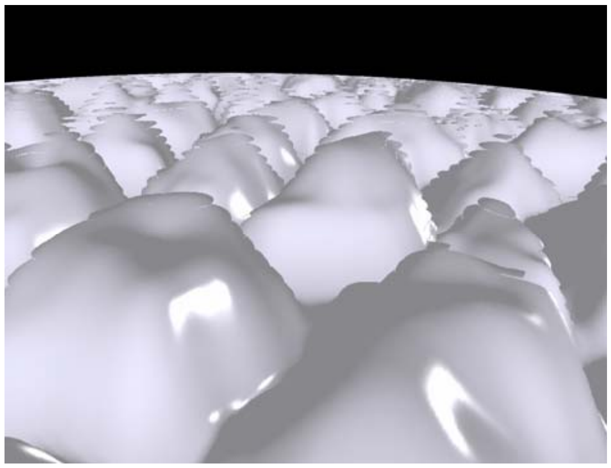


Figure 4a. *Relief mapping rendered with both linear and binary search but without depth bias applied. Notice the visual artifacts due to sampling aliasing at grazing angles.*

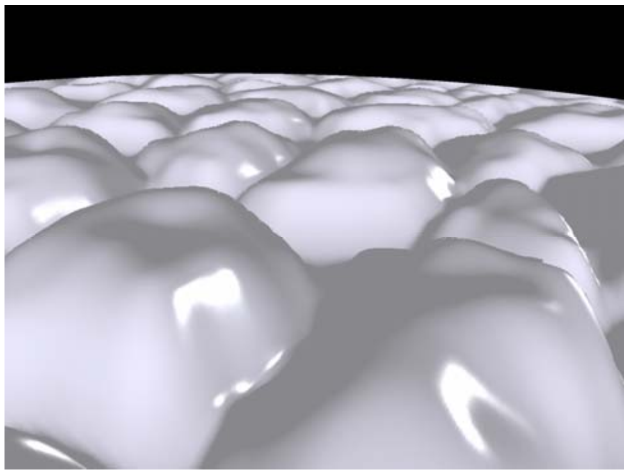


Figure 4b. *Relief mapping rendered with both linear and binary search and with depth bias applied. Notice the flattening of surface features towards the horizon.*

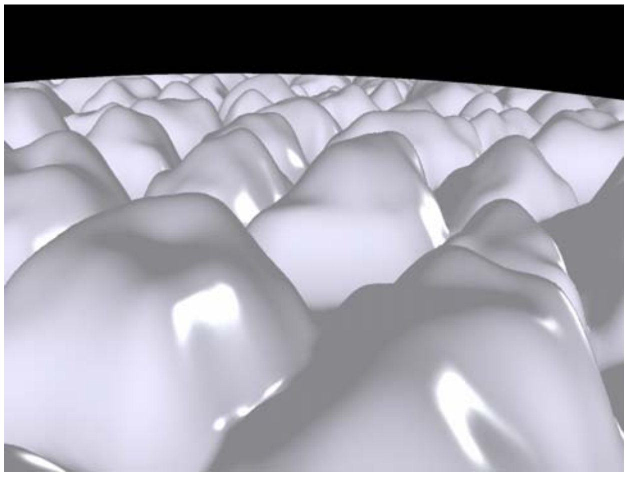


Figure 4c. *Parallax occlusion mapping rendered with the high precision height field intersection computation. Notice the lack of aliasing artifacts or feature flattening toward the horizon.*

The intersection of the surface is approximated with texture filtering, thus only using 8 bits of precision for the intersection computation. This results in visible stair-stepping artifacts at steep viewing angles (as seen in Figure 4a). Depth biasing toward the horizon hides these artifacts but introduces excessive feature flattening at oblique angles (Figure 4b).

The binary search from [Policarpo et al. 2005] requires dependent texture fetches for computation. These incur a latency cost which is not offset by any ALU computations in the relief mapping ray-height field intersection routine. Increasing the sampling rate during the binary search increases the latency of each fetch by increasing the dependency depth for each successive fetch.

Using a linear search from [Policarpo et al. 2005] without an associated binary search exacerbates the stair-stepping artifacts even with a high sampling rate (as in Figure 5a).

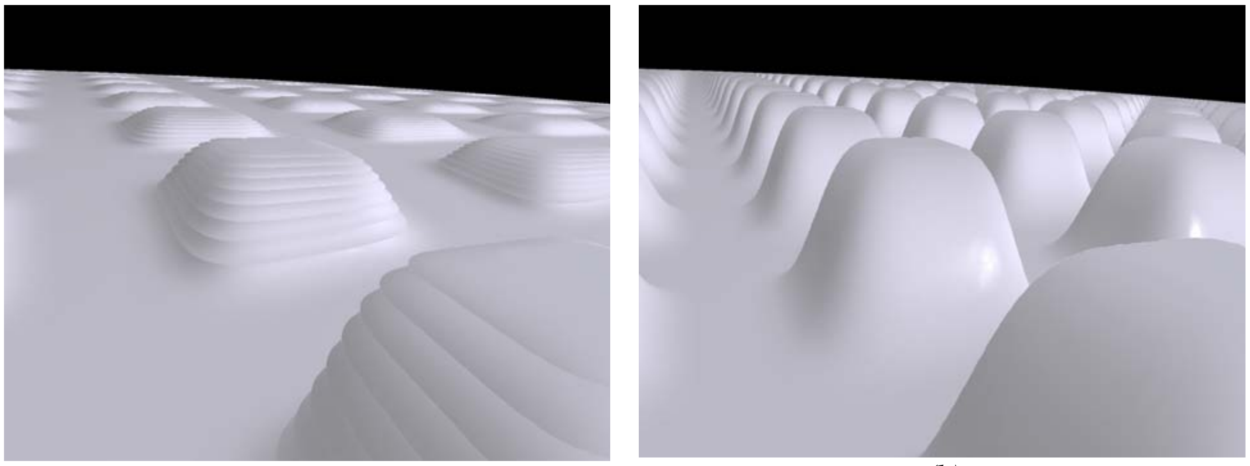


Figure 5. *Comparison of height field intersection precision using the linear search only (same assets). (a) Relief mapping. (b) Parallax occlusion mapping*

The advantage of a linear search for intersection root finding lies in an effective use of texturing hardware with low latency as it does not require dependent texture fetches. Simply using a linear search requires higher precision for the root-finding.

We sample the height field using a linear search and approximating the height profile as a piecewise linear curve (as illustrated in Figure 6). This allows us to combine the 8 bit precision due to bilinear texture filtering with the full 32 bit precision for root finding during the line intersection. Figure 4c displays the improved visual results with the lack of aliasing with using our approach.

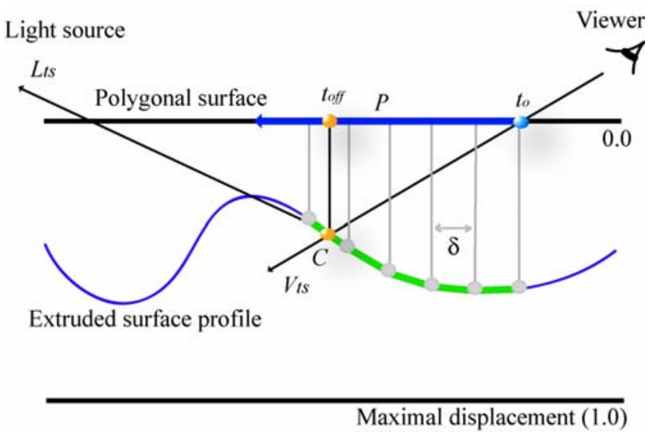


Figure 6*. We start at the input texture coordinates t_o and sample the height field profile for each linear segment of the green piecewise linear curve along parallax offset vector P. The height field profile-view ray intersection yields parallax-shifted texture coordinate offset t_{off}. δ is the interval step size. Then we perform the visibility tracing. We start at texture offset t_{off} and trace along the light direction vector L_{ts} to determine any occluding features in the height field profile.*

Since we do not encode feature information into additional look-up tables, the accuracy of our technique corresponds to the sampling interval δ (as well as for [Policarpo et al. 2005]). Both algorithms suffer from some amount of aliasing artifacts if too few samples are used for a relatively high-frequency height field, though the amount will differ between the techniques.

Automatically determining δ by using the texture resolution is currently impractical. At grazing angles, the parallax amount is quite large and thus we must march along a long parallax offset vector in order to arrive at the actual displaced point. In that case, the step size is frequently much larger than a texel, and thus unrelated to the texture resolution. To solve this, we provide both directable and automatic controls.

The artists can control the sampling size bounds with artist-editable parameters. This is convenient in real game scenarios as it allows control per texture map. If dynamic flow control (DFC) is available, we can automatically adjust the sampling rate during ray tracing. We express the sampling rate as a linear function of the angle between the geometric normal and the view direction ray:

$$n = n_{\min} + \hat{N} \bullet \hat{V}_{ts}(n_{\max} - n_{\min}) \quad (1)$$

where $n_{\min}$ and $n_{\max}$ are the artist-controlled sampling rate bounds, $\hat{N}$ is the interpolated geometric unit normal vector at the current pixel. This assures that we increase the sampling rate along the steep viewing angles. We increase the efficiency of the linear search by using the early out functionality of DFC to stop sampling the height field when a point below the surface is found.

5.4.2 Soft Shadows

The height map can in fact cast shadows on itself. This is accomplished by substituting the light vector for the view vector when computing the intersection of the height profile to determine the correct displaced texel position during the reverse height mapping step. Once we arrive at the point on the displaced surface (the point *C* in figure 6) we can compute its visibility from the any light source. For that, we cast a ray toward the light source in question and perform horizon visibility queries of the height field profile along the light direction ray $\hat{L}_{ts}$.

If there are intersections of the height field profile with $\hat{L}_{ts}$, then there are occluding features and the point in question will be in shadow. This process allows us to generate shadows due to the object features' self-occlusions and object interpenetration.

If we repeated the process for the height field profile – view direction ray tracing for the visibility query by stopping sampling at the very first intersection, we would arrive at the horizon shadowing value describing whether the displaced pixel is in shadow. Using this value during the lighting computation (as in [Policarpo et al. 2005]) generates hard shadows which can display aliasing artifacts in some scenes (Figure 7a).

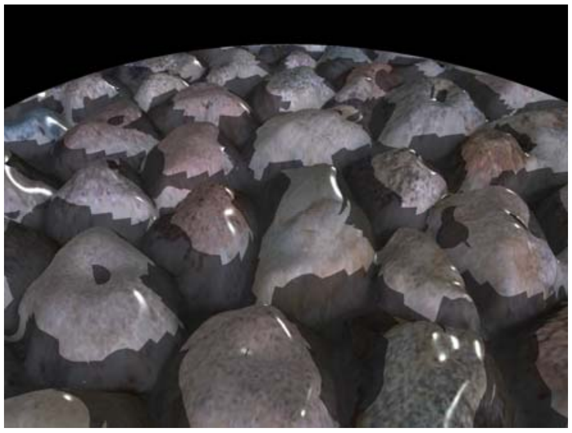


***Figure 7a**. Hard shadows generated with the relief mapping horizon visibility threshold computation.*

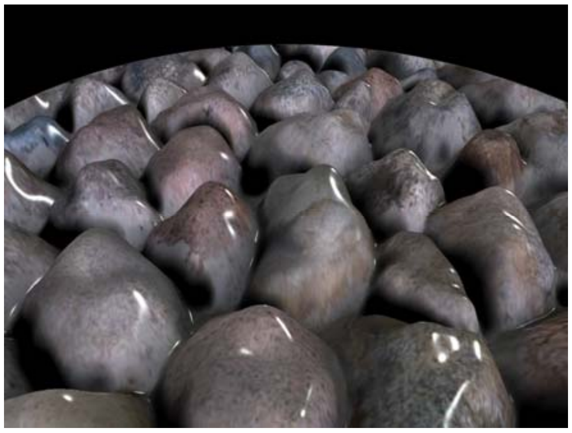


***Figure 7b.** Soft shadows generated with the parallax occlusion mapping penumbra approximation technique.*

With our approach, we sample n samples h_1 - h_n along the light direction ray (Figure 8). We assume that we are lighting the surface with an area light source and, similar to [Chan and Durand 2003] and [Wyman and Hansen 2003], we heuristically approximate the size of penumbra for the occluded pixel. Figure 9 demonstrates the penumbra size computation given an area light source, a blocker and a receiver surface.

We can use the height field profile samples h_i along the light direction ray to determine the occlusion coefficient. We sample the height value h_0 at the shifted texture coordinate t_{off}. Starting at this offset ensures that the shadows do not appear floating on top of the surface. The sample h_0 is our reference ("surface") height. We then sample n other samples along the light ray, subtracting h_0 from each of the successive samples h_i. This allows us to compute the blocker-to-receiver ratio as in figure 9. The closer the blocker is to the surface, the smaller the resulting penumbra. We compute the penumbra coefficient (the visibility coefficient v) by scaling the contribution of each sample by the distance of this sample from h_0, and using the maximum value sampled. Additionally we can weight each visibility sample to simulate the blur kernel for shadow filtering.

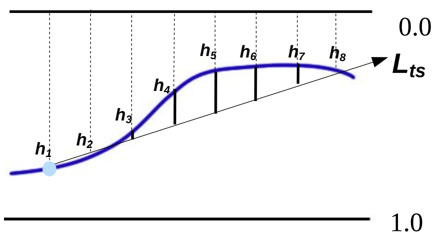


Figure 8. *Sampling the height field profile along the light ray direction L_{ts} to obtain height samples $h_1 - h_8$ (n=8)*

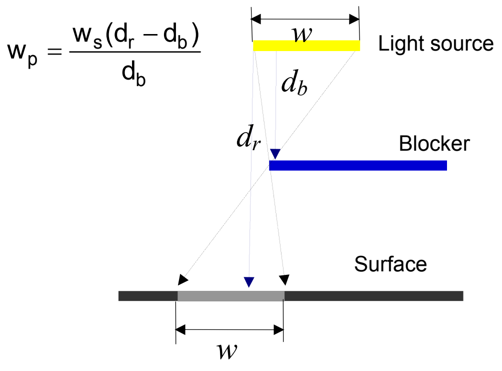


Figure 9. *Penumbra size approximation for area light sources, where w_s is the light source width, w_p is the penumbra width, d_r is the receiver depth and d_b is the blocker depth from the light source.*

We apply the visibility coefficient v during the lighting computation for generation of smooth soft shadows. This allows us to obtain well-behaved soft shadows without any edge aliasing or filtering artifacts. Figures 7b and 10 demonstrate examples of smooth shadows using our technique. One limitation of our technique is the lack of hard surface contact shadow for extremely high frequency height maps.

Remember that estimating light visibility increases shader complexity. We perform the visibility query only for areas where the dot product between the geometric normal and the light vector is non-negative by utilizing dynamic branching (see the actual pixel shader in Listing 2 in the Appendix). This allows us to compute soft shadows only for areas which are actually facing the light source.

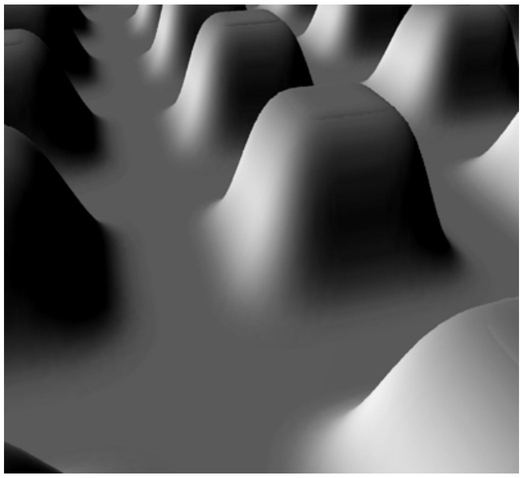


Figure 10. *Smooth soft shadows and perspective-correct depth details generated with the parallax occlusion rendering algorithm*

5.4.3 Adaptive Level-of-Detail Control System

We compute the current mip level explicitly in the pixel shader and use this information to transition between different levels of detail from the full effect to bump mapping. Simply using mip-mapping for LOD management is ineffective since it does not reduce shader complexity during rendering. Using the full shader for the height field profile intersection with the view ray and the light ray, the visibility and lighting is expensive. Although at lower mip levels the fill is reduced, without our level-of-detail system, the full shader will be executed for each pixel regardless of its proximity to the viewer. Instead, with our algorithm, only a simple bump mapping shader is executed for mip levels higher than the specified threshold value.

This in-shader LOD system provides a significant rendering optimization and smooth transitions between the full parallax occlusion mapping and a simplified representation without visual artifacts such as ghosting or popping. Since all calculations are performed per pixel, the method robustly handles extreme close-ups of the object surface, thus providing an additional level of detail management.

We compute the mip level directly in the pixel shader (as described in [Shreiner et al. 2005]) on SM 3.0 hardware (see the actual pixel shader in Listing 2 in the Appendix). The lowest level of detail is rendered using bump mapping. As we determine that the currently rendered level of detail is close enough to the threshold, we interpolate the parallax occlusion-mapped lit result with the bump-mapped lit result using the fractional part of the current mip level as the interpolation parameter. There is almost no associated visual quality degradation as we move into a lower level of detail and the transition appears quite smooth (Figure 11).

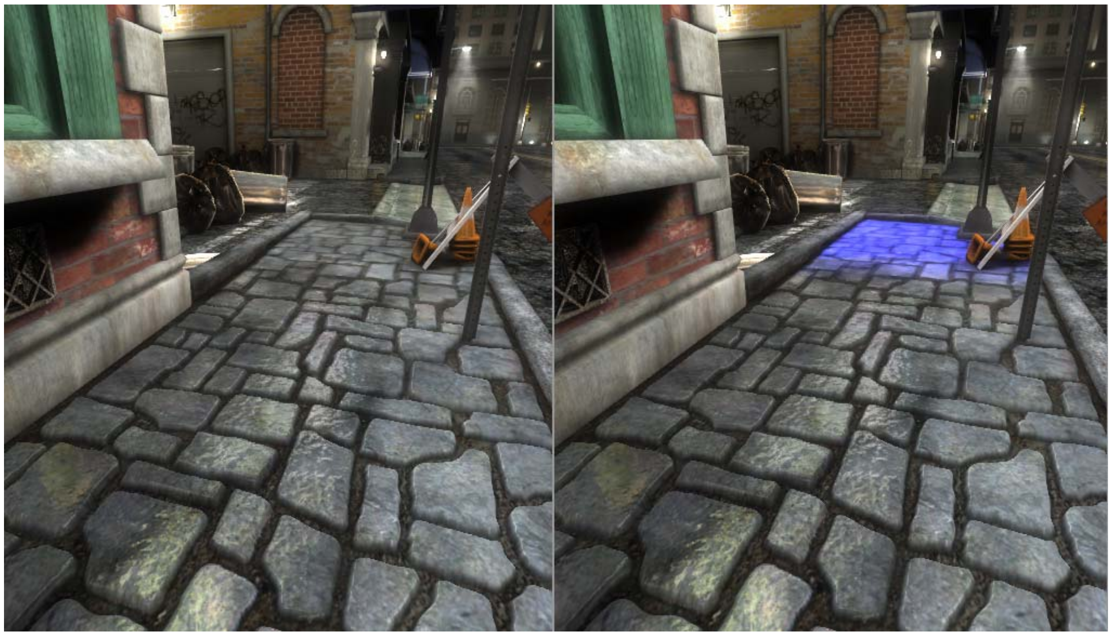


Figure 11. *A complex scene with parallax occlusion mapping on the sidewalk and the brick wall. The strength of the blue tint in (b) denotes the decreasing level of detail (deepest blue being bump mapping and no blue displays full computation). Note the lack of visual artifacts and smooth transition due to the level-of-detail transition discernable in (a). The transition region is very small.*

We expose the threshold level parameter to the artists in order to provide directability for game level editing. In our examples we used a threshold value of 4. Thus even at steep grazing angles the close-up views of surfaces will maintain perspective correct depth.

5.5 Results

We have implemented the techniques described in this paper using DirectX 9.0c shader programs on a variety of graphics cards. An example of an educational version of this shader is shown in listings 1 (for the vertex shader implementation using DirectX 9.0c shader model 3.0) and listing 2 (for the pixel shader implementation using DirectX 9.0c shader model 3.0). We use different texture sizes selected based on the desired feature resolution. For Figures 1, 2, 11, 16, and 17 we apply 1024x1024 RGBα textures with non-contiguous texture coordinates. For Figures 4 and 7 we apply repeated 256x256 RGBα textures, and for Figures 5 and 10 we use repeated 128x128 RGBα textures.

We applied parallax occlusion mapping to a 1,100 polygon soldier character shown in figure 12a. We compared this result to a 1.5 million polygon version of the soldier model used to generate normal maps for the low resolution version (Figure 12b). (Note that the two images in figure 12 are from slightly different viewpoints though extracted from a demo sequence with similar viewpoint paths.) We apply a 2048x2048 RGBα texture map to the low resolution object. We render the low resolution soldier using DirectX on ATI

Radeon X1600 XL at 255 fps. The sampling rate bounds were set to the range of [8, 50] range. The memory requirement for this model was 79K for the vertex buffer, 6K for the index buffer, and 13MB of texture memory (using 3DC texture compression).

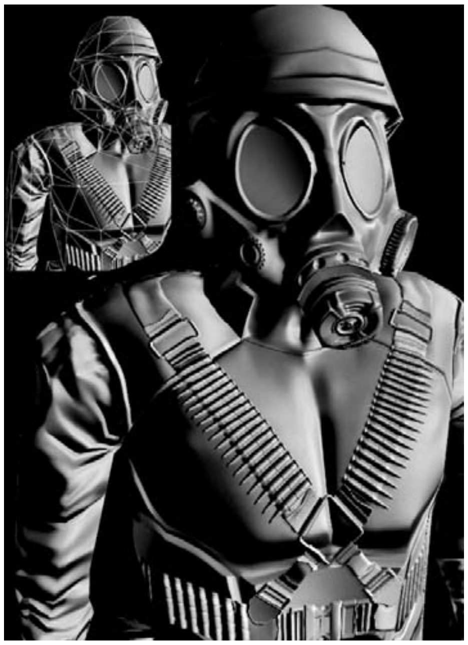


Figure 12a. *An 1,100 polygon game soldier character with parallax occlusion mapping*

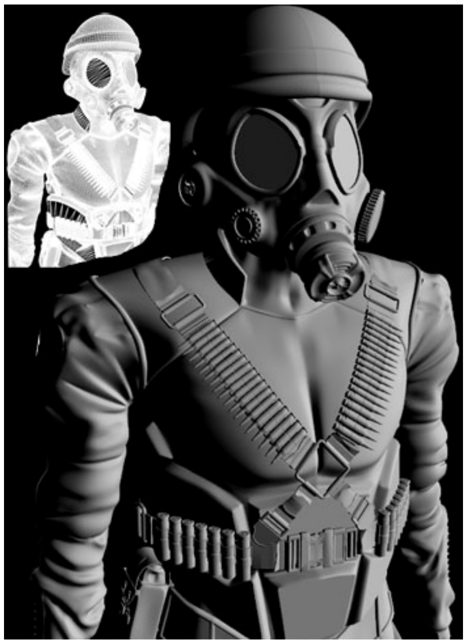


Figure 12b. *A 1.5 million polygon soldier character with diffuse lighting*

The high resolution soldier model is rendered on the same hardware at a rate of 32 fps. The memory requirement for this model was 31MB for the vertex buffer and 14MB for the index buffer. Due to memory considerations, vertex transform cost for rendering, animation, and authoring issues, characters matching the high resolution soldier are impractical in current game scenarios. However, using our technique on an extremely low resolution model provided significant frame rate increase with 32MB of memory being saved, at a comparable quality of rendering.

This demonstrates the usefulness of the presented technique for texture-space displacement mapping via parallax occlusion mapping. In order to render the same objects interactively with equal level of detail, the meshes would need an extremely detailed triangle subdivision, which is impractical even with the currently available GPUs.

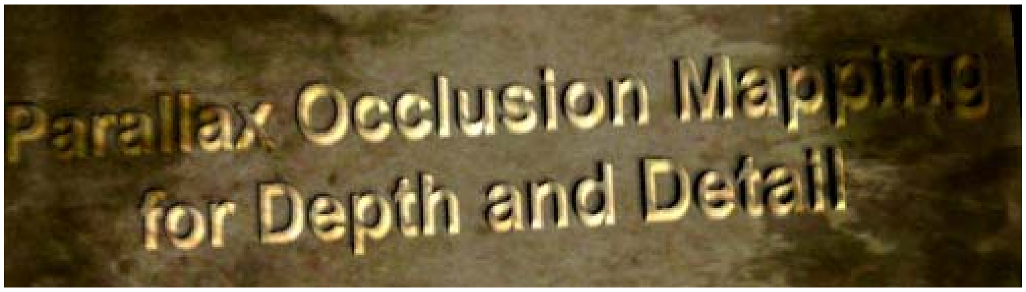
Our method can be used with a dynamically rendered height field and still produce perspective-correct depth results. In that case, the dynamically updated displacement values can be used to derive the normal vectors at rendering time by convolving the height map with a Sobel operator in the horizontal and vertical direction (as described in detail in [Tatarchuk06a]). The rest of the algorithm does not require any modification.

We used this technique extensively in the interactive demo called “ToyShop” [Toyshop05] for a variety of surfaces and effects. As seen in Figure 1 and 16, we’ve rendered the cobblestone sidewalk using this technique (using sampling range from 8 to 40 samples per pixel), in Figure 13 we have applied it to the brick wall (with the same sampling range), and in Figure 17 we see parallax occlusion mapping used to render extruded wood-block letters of the ToyShop store sign. We were able to integrate a variety of lighting models with this technique, ranging from a single diffusely lit material in Figure 13, to shadows and specular illumination in Figure 17, and shadow mapping integrated and dynamic view-dependent reflections in Figure 1a.

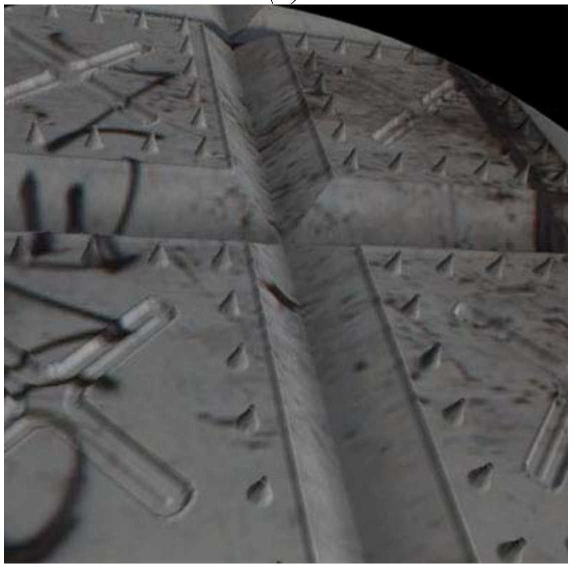
5.6 Considerations for practical use of parallax occlusion mapping and game integration

5.6.1 Algorithm limitations and relevant considerations

Although parallax occlusion mapping is a very powerful and flexible technique for computing and lighting extruded surfaces in real-time, it does have its limitations. Parallax occlusion mapping is a sampling-based algorithm at its core, and as such, it can exhibit aliasing. The frequencies of the height field will determine the required sampling rate for the ray tracing procedures – otherwise aliasing artifacts will be visible (as seen in Figures 4a and 5a). One must increase the sampling rate significantly if the height field contains very sharp features (as visible in the text and sharp conic features in Figure 13 below). However, as we can note from the images in Figures 13a and 13b, the visual quality of the results rendered with parallax occlusion mapping is high enough to render such traditionally difficult objects as extruded text or sharp peaks at highly interactive rates (fps > 15fps on ATI Radeon X1600 XL rendering at 1600x1200 resolution). In order to render the same objects interactively with equal level of detail, the meshes would need an extremely detailed triangle subdivision (with triangles being nearly pixel-sized), which is impractical even with the currently available GPUs.



(a)



(b)

Figure 13. *Rendering extruded text objects in (a) and sharp conic features in (b) with parallax occlusion mapping*

The sampling limitation is particularly evident in the DirectX 9.0c shader model 2.0 implementation of the parallax occlusion mapping algorithm if the height field used has high spatial frequency content. This specific shader model suffers from a small instruction count limit, and thus we are unable to compute more than 8 samples during ray tracing in a single pass. However, several passes can be used to compute the results of ray tracing by using offscreen buffer rendering to increase the resulting precision of computations using SM 2.0 shaders. As in the analog-to-digital sound conversion process, sampling during the ray tracing at slightly more than twice the frequency of the height map features will make up for not modeling the surfaces with implicit functions and performing the exact intersection of the ray with the implicit representation of the extruded surface.

Another limitation of our technique is the lack of detailed silhouettes, betraying the low resolution geometry actually rendered with our method. This is an important feature for enhancing the realism of the effect and we are investigating ideas for generating correct silhouettes. However, in many scenarios, the artists can work around this issue by placing specific ‘border’ geometry to hide the artifacts. One can notice this at work in the “ToyShop” demo as the artists placed the curb stones at the edge of the sidewalk object with parallax occlusion mapped cobblestones or with a special row of bricks at the corner of the brick building seen in Figure 14 below.

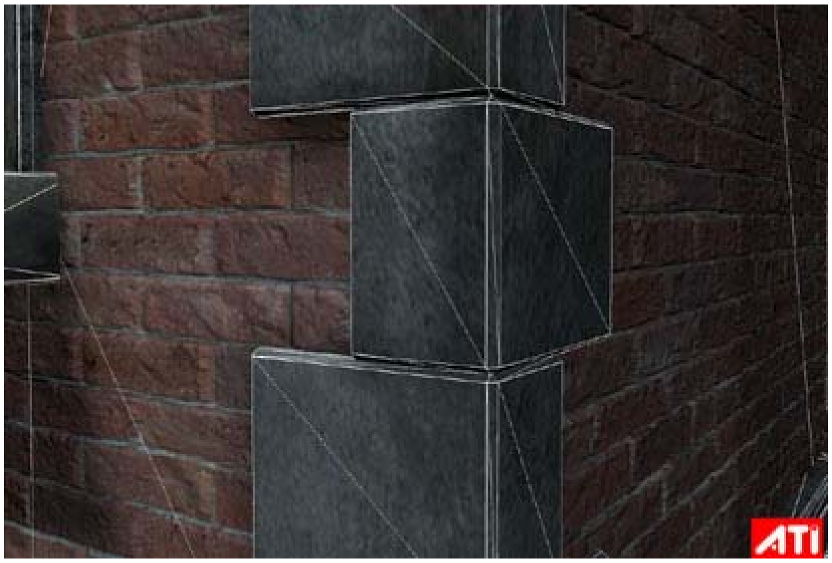


Figure 14. *Additional brick geometry placed at the corners of parallax occlusion mapped brick objects to hide inaccurate silhouettes in an interactive environment of the "ToyShop" demo.*

The parallax occlusion mapping algorithm will not automatically produce surfaces with correct depth buffer values (since it simply operates in screen-space on individual pixels). This means that in some situations this will result in apparent object interpenetration or incorrect object collision. The algorithm can be extended to output accurate depth quite easily. Since we know the reference surface's geometric depth, we can compute the displacement amount by sampling the height field at the t_{off} location and adding or subtracting this displacement amount to the reference depth value (as described in [Policarpo05]) by outputting it as the depth value from the pixel shader.

5.6.1 Art Content Authoring Suggestions for Parallax Occlusion Mapping

Adding art asset support for parallax occlusion mapping requires a minimal increase in memory footprint (for an additional 8-bit height map) if the application already supports normal mapping and contains appropriate assets. There are many reliable methods for generating height maps useful for this technique:

- Normal maps can be generated from a combination of a low- and high-resolution models with the *NormalMapper* software [NormalMapper03]. The tool has an option to simultaneously output the corresponding displacement values in a height map
- A height map may be painted in a 3D painting software like ZBrush^{TM}
- It also can be created in a 2D painting software such as Adobe® Photoshop^{TM}

The parallax occlusion mapping technique is an efficient and compelling technique for simulating surface details. However, as with other bump mapping techniques, its quality depends strongly on the quality of its art content. Empirically, we found that lower-frequency height map textures result in higher performance (due to less samples required for ray tracing) and better quality results (since less height field features are missed). For example, if creating height maps for rendering bricks or cobblestones, one may widen the grout region and apply a soft blur to smooth the transition and thus lower the height map frequency content. As discussed in the previous section, when using high frequency height maps (such as those in Figure 13a or 13b), we must increase the range of sampling for ray tracing.

An important consideration for authoring art assets for use with this algorithm lies in the realization that the algorithm always extrudes surfaces “pushing down” – unlike the traditional displacement mapping. This affects the placement of the original low resolution geometry – the surfaces must be placed slightly higher than where the anticipated extruded surface should be located. Additionally this means that the peaks in the extruded surface will correspond to the brightest values in the height map (white) and the valleys will be corresponding to the darkest (black).

5.7 Conclusions

We have presented a pixel-driven displacement mapping technique for rendering detailed surfaces under varying light conditions, generating soft shadows due to self-occlusion. We have described an efficient algorithm for computing intersections of the height field profile with rays with high precision. Our method includes estimation of light visibility for generation of soft shadows. An automatic level-of-detail control system manages shader complexity efficiently at run-time, generating smooth LOD transitions without visual artifacts. Our technique takes advantage of the programmable GPU pipeline resulting in highly interactive frame rates coupled with a low memory footprint.

Parallax occlusion mapping can be used effectively to generate an illusion of very detailed geometry exhibiting correct motion parallax as well as producing very convincing self-shadowing effects. We provide a high level of directability to the artists and significantly improved visual quality over the previous approaches. We hope to see more games implementing compelling scenes using this technique.

5.7 Acknowledgements

We would like to thank Dan Roeger, Daniel Szecket, Abe Wiley and Eli Turner for their help with the artwork, and Zoë Brawley from Relic Entertainment for her ideas in the original implementation of the 2004 technique. We also would like to thank Pedro Sander, John Isidoro, Thorsten Scheuermann, and Chris Oat of ATI Research, Inc., Jason L. Mitchell of Valve Software, and Eric Haines, of Autodesk for their help, suggestions and review of this work.

5.8 Bibliography

BECKER, B. G., AND MAX, N. L. 1993. Smooth Transitions between Bump Rendering Algorithms. In ACM Transactions on Graphics (Siggraph 1993 Proceedings), ACM Press, pp. 183-190

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.

BRAWLEY, Z., AND TATARCHUK, N. 2004. Parallax Occlusion Mapping: Self-Shadowing, Perspective-Correct Bump Mapping Using Reverse Height Map Tracing. In ShaderX^3: Advanced Rendering with DirectX and OpenGL, Engel, W., Ed., Charles River Media, pp. 135-154.

CHAN, E., AND DURAND, F. 2003. Rendering fake soft shadows with smoothies, In Eurographics Symposium on Rendering Proceedings, ACM Press, pp. 208-218.

COOK, R. L. 1984. Shade Trees, In Proceedings of the 11th annual conference on Computer graphics and interactive techniques, ACM Press, pp. 223-231.

DOGGETT, M., AND HIRCHE, J. 2000. Adaptive View Dependent Tessellation of Displacement Maps. In HWWS ’00: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics Hardware, ACM Press, pp. 59-66.

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

FOURNIER, A. 1992. Filtering Normal Maps and Creating Multiple Surfaces, Technical Report, University of British Columbia.

HEIDRICH, W., AND SEIDEL, H.-P. 1998. Ray-tracing Procedural Displacement Shaders, In Graphics Interface, pp. 8-16.

HIRCHE, J., EHLERT, A., GUTHE, S., DOGGETT, M. 2004. Hardware Accelerated Per-Pixel Displacement Mapping. In Graphics Interface, pp. 153-158.

KANEKO, T., TAKAHEI, T., INAMI, M., KAWAKAMI, N., YANAGIDA, Y., MAEDA, T., TACHI, S. 2001. Detailed Shape Representation with Parallax Mapping. In Proceedings of ICAT 2001, pp. 205-208.

KAUTZ, J., AND SEIDEL, H.-P. 2001. Hardware accelerated displacement mapping for image based rendering. In Proceedings of Graphics Interface 2001, B.Watson and J.W. Buchanan, Eds., pp. 61–70.

MAX, N. 1988. Horizon mapping: shadows for bump-mapped surfaces. The Visual Computer 4, 2, pp. 109–117.

MCGUIRE, M. AND MCGUIRE, M. 2005. Steep Parallax Mapping. I3D 2005 Poster.

AKENINE-MÖLLER, T., HEINES, E. 2002. *Real-Time Rendering*, 2nd Edition, A.K. Peters, July 2002

OLANO, M., KUEHNE, B., SIMMONS, M. 2003. Automatic Shader Level of Detail. In Siggraph/Eurographics Workshop on Graphics Hardware Proceedings, ACM Press, pp. 7-14.

OLIVEIRA, M. M, AND POLICARPO, F.. 2005. An Efficient Representation for Surface Details. UFRGS Technical Report RP-351.

OLIVEIRA, M. M., BISHOP, G., AND MCALLISTER, D. 2000. Relief texture mapping. In Siggraph 2000, Computer Graphics Proceedings, ACM Press / ACM SIGGRAPH / Addison Wesley Longman, K. Akeley, Ed., pp. 359–368.

PHARR, M., AND HANRAHAN, P. 1996. Geometry caching for ray-tracing displacement maps. In Eurographics Rendering Worshop 1996, Springer Wien, New York City, NY, X. Pueyo and P. Schröder, Eds., pp. 31–40.

POLICARPO, F., OLIVEIRA, M. M., COMBA, J. 2005. Real-Time Relief Mapping on Arbitrary Polygonal Surfaces. In ACM SIGGRAPH Symposium on Interactive 3D Graphics Proceedings, ACM Press, pp. 359-368.

POLICARPO, F., OLIVEIRA, M. M. 2006. Relief Mapping of Non-Height-Field Surface Details. In ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games Proceedings, ACM Press, pp. 55-52.

SHREINER, D., WOO, M., NEIDER, J., DAVIS, T.. 2005. OpenGL® Programming Guide: The Official Guide to Learning OpenGL®, version 2, Addison-Wesley.

SLOAN, P-P. J., AND COHEN, M. F. 2000. Interactive Horizon Mapping. In 11th Eurographics Workshop on Rendering Proceedings, ACM Press, pp. 281-286.

TATARCHUK, N. 2006. Dynamic Parallax Occlusion Mapping with Approximate Soft Shadows. In the proceedings of ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games, pp. 63-69

TATARCHUK, N. 2006a. Practical Parallax Occlusion Mapping for Highly Detailed Surface Rendering. In the proceedings of Game Developer Conference

WANG, L., WANG, X., TONG, X., LIN, S., HU, S., GUO, B., AND SHUM, H.-Y. 2003. View-dependent displacement mapping. ACM Trans. Graph. 22, 3, pp. 334–339.

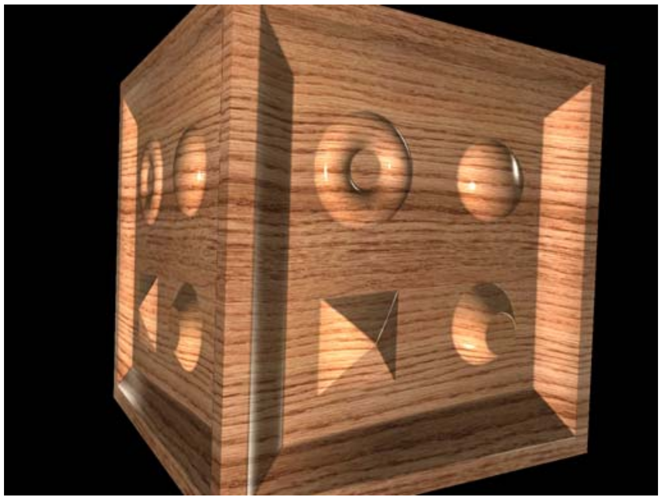
WANG, X., TONG, X., LIN, S., HU, S., GUO, B., AND SHUM, H.-Y. 2004. Generalized displacement maps. In Eurographics Symposium on Rendering 2004, EUROGRAPHICS, Keller and Jensen, Eds., EUROGRAPHICS, pp. 227–233.

WELSH, T. 2004. Parallax Mapping, ShaderX3: Advanced Rendering with DirectX and OpenGL, Engel, W. Ed., A.K. Peters, 2004

WYMAN, C., AND HANSEN, C. 2002. Penumbra maps: approximate soft shadows in real-time. In Eurographics workshop on Rendering 2003, EUROGRAPHICS, Keller and Jensen, Eds., EUROGRAPHICS, pp. 202-207.

TOYSHOP DEMO, 2005. ATI Research, Inc. Can be downloaded from http://www.ati.com/developer/demos/rx1800.html

NORMALMAPPER TOOL, 2003. ATI Research, Inc. Can be downloaded from http://www2.ati.com/developer/NormalMapper-3_2_2.zip


(a)

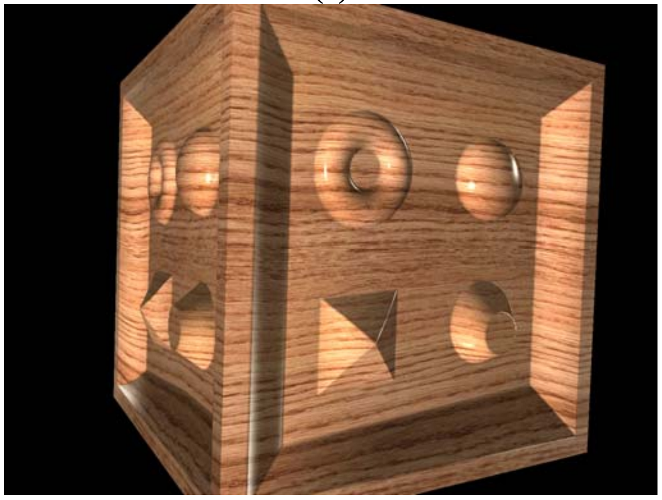

(b)

Figure 15. *Simple cube model rendered with detailed surface detailed from the same viewpoint. In (a), relief mapping is used to create surface complexity. In (b), parallax occlusion mapping is used to render perspective-correct extruded surfaces. Notice the differences on the left face of the cube as the surface is viewed at a steep angle.*

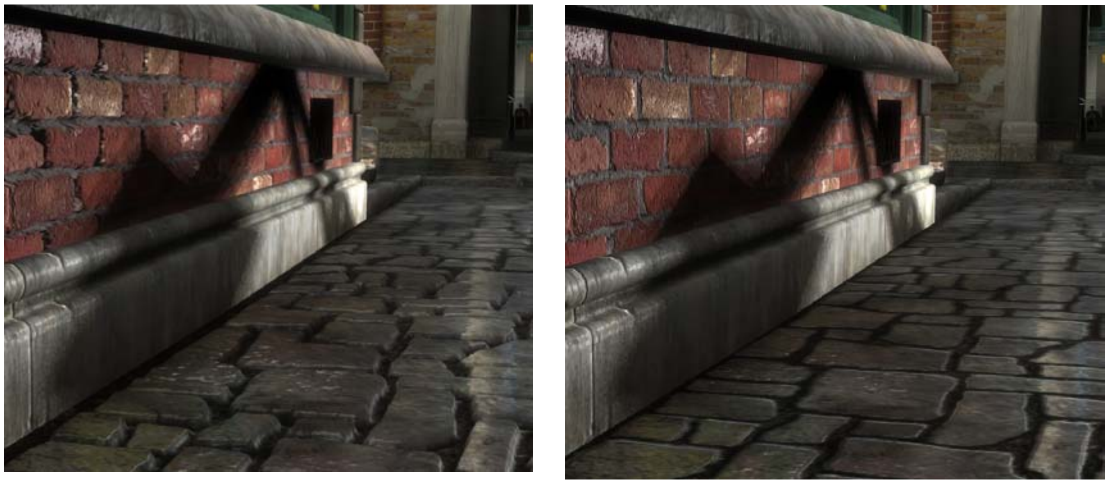


Figure 16. *A portion of a realistic city environment with the cobblestone sidewalk and the brick wall rendered with parallax occlusion mapping (left) and bump mapping (right). We are able to use shadow mapping on the surfaces and dynamically rendered reflections from objects in the scene.*

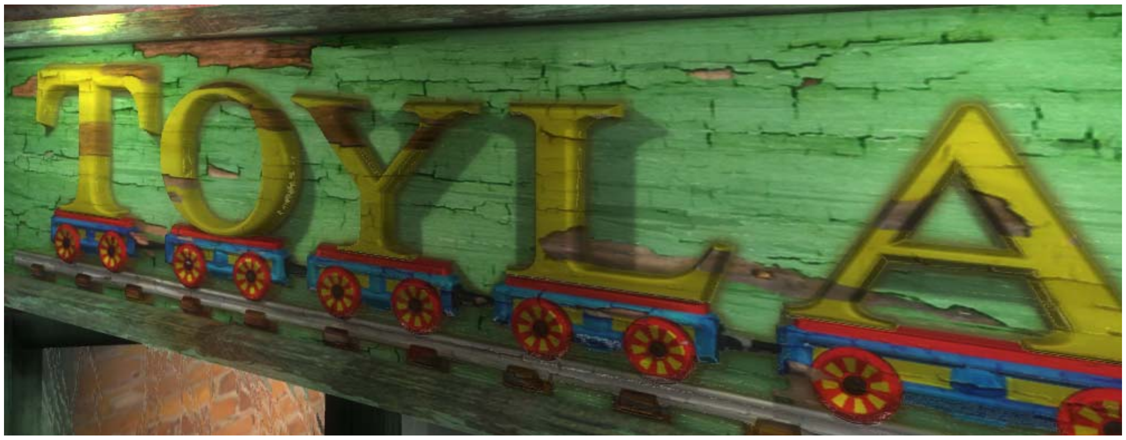


Figure 17. *Displaced text rendering with the sign rendered using parallax occlusion mapping technique*

Appendix. DirectX shader code implementation of Parallax Occlusion Mapping.

```
float4x4 matViewInverse;
float4x4 matWorldViewProjection;
float4x4 matView;

float    fBaseTextureRepeat;
float    fHeightMapRange;
float4   vLightPosition;

struct VS_INPUT
{
   float4 positionWS  : POSITION;
   float2 texCoord    : TEXCOORD0;
   float3 vNormalWS   : NORMAL;
   float3 vBinormalWS : BINORMAL;
   float3 vTangentWS  : TANGENT;
};

struct VS_OUTPUT
{
   float4 position : POSITION;
   float2 texCoord : TEXCOORD0;

   // Light vector in tangent space, not normalized
   float3 vLightTS : TEXCOORD1;

   // View vector in tangent space, not normalized
   float3 vViewTS : TEXCOORD2;

   // Parallax offset vector in tangent space
   float2 vParallaxOffsetTS : TEXCOORD3;

   // Normal vector in world space
   float3 vNormalWS : TEXCOORD4;

   // View vector in world space
   float3 vViewWS : TEXCOORD5;
};

VS_OUTPUT vs_main( VS_INPUT i )
{
   VS_OUTPUT Out = (VS_OUTPUT) 0;

   // Transform and output input position
   Out.position = mul( matWorldViewProjection, i.positionWS );

   // Propagate texture coordinate through:
   Out.texCoord = i.texCoord;

   // Uncomment this to repeat the texture
   // Out.texCoord *= fBaseTextureRepeat;

   // Propagate the world vertex normal through:
```

```
   Out.vNormalWS = i.vNormalWS;

   // Compute and output the world view vector:
   float3 vViewWS = mul( matViewInverse,
                    float4(0,0,0,1)) - i.positionWS;

   Out.vViewWS = vViewWS;

   // Compute denormalized light vector in world space:
   float3 vLightWS = vLightPosition - i.positionWS;

   // Normalize the light and view vectors and transform
   // it to the tangent space:
   float3x3 mWorldToTangent =
      float3x3( i.vTangentWS, i.vBinormalWS, i.vNormalWS );

   // Propagate the view and the light vectors (in tangent space):
   Out.vLightTS = mul( mWorldToTangent, vLightWS );
   Out.vViewTS  = mul( mWorldToTangent, vViewWS  );

   // Compute the ray direction for intersecting the height field
   // profile with current view ray. See the above paper for derivation
   // of this computation.

   // Compute initial parallax displacement direction:
   float2 vParallaxDirection = normalize(  Out.vViewTS.xy );

   // The length of this vector determines the furthest amount
   // of displacement:
   float fLength = length( Out.vViewTS );
   float fParallaxLength = sqrt( fLength * fLength - Out.vViewTS.z
            * Out.vViewTS.z ) / Out.vViewTS.z;

   // Compute the actual reverse parallax displacement vector:
   Out.vParallaxOffsetTS = vParallaxDirection * fParallaxLength;

   // Need to scale the amount of displacement to account for
   // different height ranges in height maps. This is controlled by
   // an artist-editable parameter:
   Out.vParallaxOffsetTS *= fHeightMapRange;

   return Out;

}   // End of VS_OUTPUT vs_main(..)
```

Listing 1. *Parallax occlusion mapping algorithm implementation. Vertex shader, DirectX 9.0c shader model 3.0*

```
// NOTE: Since for this particular example want to make convenient ways
// to turn features rendering on and off (for example, for turning on /
// off visualization of current level of details, shadows, etc), the
// shader presented uses extra flow control instructions than it would
// in a game engine.

// Uniform shader parameters declarations
bool bVisualizeLOD;
bool bVisualizeMipLevel;
bool bDisplayShadows;

// This parameter contains the dimensions of the height map / normal map
// pair and is used for determination of current mip level value:
float2 vTextureDims;

int   nLODThreshold;
float fShadowSoftening;
float fSpecularExponent;
float fDiffuseBrightness;
float fHeightMapRange;

float4 cAmbientColor;
float4 cDiffuseColor;
float4 cSpecularColor;

int nMinSamples;
int nMaxSamples;

sampler tBaseMap;
sampler tNormalMap;

// Note: centroid sampling should be specified if multisampling is
// enabled
struct PS_INPUT
{
   float2 texCoord : TEXCOORD0;

   // Light vector in tangent space, denormalized
   float3 vLightTS : TEXCOORD1_centroid;

   // View vector in tangent space, denormalized
   float3 vViewTS : TEXCOORD2_centroid;

   // Parallax offset vector in tangent space
   float2 vParallaxOffsetTS : TEXCOORD3_centroid;

   // Normal vector in world space
   float3 vNormalWS : TEXCOORD4_centroid;

   // View vector in world space
   float3 vViewWS : TEXCOORD5_centroid;
};
```

```
//...........................................................................
// Function:    ComputeIllumination
//
// Description: Computes phong illumination for the given pixel using
//              its attribute textures and a light vector.
//...........................................................................
float4 ComputeIllumination( float2 texCoord, float3 vLightTS,
                            float3 vViewTS,  float  fOcclusionShadow )
{
   // Sample the normal from the normal map for the given texture sample:
   float3 vNormalTS = normalize( tex2D( tNormalMap, texCoord ) * 2 - 1 );

   // Sample base map:
   float4 cBaseColor = tex2D( tBaseMap, texCoord );

   // Compute diffuse color component:
   float4 cDiffuse = saturate( dot( vNormalTS, vLightTS )) *
                               cDiffuseColor;

   // Compute specular component:
   float3 vReflectionTS = normalize( 2 * dot( vViewTS, vNormalTS ) *
                                     vNormalTS - vViewTS );

   float fRdotL = dot( vReflectionTS, vLightTS );

   float4 cSpecular = saturate( pow( fRdotL, fSpecularExponent )) *
                      cSpecularColor;

   float4 cFinalColor = (( cAmbientColor + cDiffuse ) * cBaseColor +
                      cSpecular ) * fOcclusionShadow;

   return cFinalColor;
}

//............................................................................
// Function:    ps_main
//
// Description: Computes pixel illumination result due to applying
//              parallax occlusion mapping to simulation of view-
//              dependent surface displacement for a given height map
//............................................................................
float4 ps_main( PS_INPUT i ) : COLOR0
{
   //  Normalize the interpolated vectors:
   float3 vViewTS   = normalize( i.vViewTS  );
   float3 vViewWS   = normalize( i.vViewWS  );
   float3 vLightTS  = normalize( i.vLightTS );
   float3 vNormalWS = normalize( i.vNormalWS );

   float4 cResultColor = float4( 0, 0, 0, 1 );

   // Adaptive in-shader level-of-detail system implementation.
   // Compute the current mip level explicitly in the pixel shader
   // and use this information to transition between different levels
   // of detail from the full effect to simple bump mapping.

   // Compute the current gradients:
```

```
   float2 fTexCoordsPerSize = i.texCoord * vTextureDims;

   // Compute all 4 derivatives in x and y in a single instruction
   //  to optimize:
   float2 dxSize, dySize;
   float2 dx, dy;

   float4( dxSize, dx ) = ddx( float4( fTexCoordsPerSize, i.texCoord ) );
   float4( dySize, dy ) = ddy( float4( fTexCoordsPerSize, i.texCoord ) );

   float  fMipLevel;
   float  fMipLevelInt;    // mip level integer portion
   float  fMipLevelFrac;   // mip level fractional amount for
                           // blending in between levels

   float  fMinTexCoordDelta;
   float2 dTexCoords;

   // Find min of change in u and v across quad: compute du and dv
   // magnitude across quad
   dTexCoords = dxSize * dxSize + dySize * dySize;

   // Standard mipmapping uses max here
   fMinTexCoordDelta = max( dTexCoords.x, dTexCoords.y );

   // Compute the current mip level  (* 0.5 is effectively
   // computing a square root before )
   fMipLevel = max( 0.5 * log2( fMinTexCoordDelta ), 0 );

   // Start the current sample located at the input texture
   // coordinate, which would correspond to computing a bump
   // mapping result:
   float2 texSample = i.texCoord;

   // Multiplier for visualizing the level of detail
   float4 cLODColoring = float4( 1, 1, 3, 1 );

   float fOcclusionShadow = 1.0;

   if ( fMipLevel <= (float) nLODThreshold )
   {
      //===============================================//
      // Parallax occlusion mapping offset computation //
      //===============================================//

      // Utilize dynamic flow control to change the number of samples
      // per ray depending on the viewing angle for the surface.
      // Oblique angles require smaller step sizes to achieve
      // more accurate precision for computing displacement.
      // We express the sampling rate as a linear function of the
      // angle between the geometric normal and the view direction ray:
      int nNumSteps = (int) lerp( nMaxSamples, nMinSamples,
                                  dot( vViewWS, vNormalWS ) );

      // Intersect the view ray with the height field profile along
      // the direction of the parallax offset ray (computed in the
      // vertex shader. Note that the code is designed specifically
      // to take advantage of the dynamic flow control constructs in HLSL
      // and is very sensitive to the specific language syntax.
      // When converting to other examples, if still want to use dynamic
      // flow control in the resulting assembly shader, care must be
      // applied.
      // In the below steps we approximate the height field profile
      // as piecewise linear curve. We find the pair of endpoints
      // between which the intersection between the height field
      // profile and the view ray is found and then compute line segment
      // intersection for the view ray and the line segment formed by
      // the two endpoints. This intersection is the displacement
      // offset from the original texture coordinate.

      float fCurrHeight = 0.0;
      float fStepSize   = 1.0 / (float) nNumSteps;
      float fPrevHeight = 1.0;
      float fNextHeight = 0.0;

      int    nStepIndex = 0;
      bool   bCondition = true;

      float2 vTexOffsetPerStep = fStepSize * i.vParallaxOffsetTS;
      float2 vTexCurrentOffset = i.texCoord;
      float  fCurrentBound     = 1.0;
      float  fParallaxAmount   = 0.0;

      float2 pt1 = 0;
      float2 pt2 = 0;

      float2 texOffset2 = 0;

      while ( nStepIndex < nNumSteps )
      {
         vTexCurrentOffset -= vTexOffsetPerStep;

         // Sample height map which in this case is stored in the
         // alpha channel of the normal map:
         fCurrHeight = tex2Dgrad( tNormalMap, vTexCurrentOffset,
                                  dx, dy ).a;

         fCurrentBound -= fStepSize;

         if ( fCurrHeight > fCurrentBound )
         {
            pt1 = float2( fCurrentBound, fCurrHeight );
            pt2 = float2( fCurrentBound + fStepSize, fPrevHeight );

            texOffset2 = vTexCurrentOffset - vTexOffsetPerStep;

            nStepIndex = nNumSteps + 1;
         }
         else
         {
            nStepIndex++;
            fPrevHeight = fCurrHeight;
         }
      }   // End of while ( nStepIndex < nNumSteps )
      float fDelta2 = pt2.x - pt2.y;
      float fDelta1 = pt1.x - pt1.y;
      fParallaxAmount = (pt1.x * fDelta2 - pt2.x * fDelta1 ) /
                        ( fDelta2 - fDelta1 );
      float2 vParallaxOffset = i.vParallaxOffsetTS *
                        (1 - fParallaxAmount );

      // The computed texture offset for the displaced point
      // on the pseudo-extruded surface:
      float2 texSampleBase = i.texCoord - vParallaxOffset;
      texSample = texSampleBase;

      // Lerp to bump mapping only if we are in between,
      // transition section:
      cLODColoring = float4( 1, 1, 1, 1 );

      if ( fMipLevel > (float)(nLODThreshold - 1) )
      {
         // Lerp based on the fractional part:
         fMipLevelFrac = modf( fMipLevel, fMipLevelInt );

         if ( bVisualizeLOD )
         {
            // For visualizing: lerping from regular POM-
            // resulted color through blue color for transition layer:
            cLODColoring = float4( 1, 1, max(1, 2 * fMipLevelFrac), 1 );
         }

         // Lerp the texture coordinate from parallax occlusion
         // mapped coordinate to bump mapping smoothly based on
         // the current mip level:
         texSample = lerp( texSampleBase, i.texCoord, fMipLevelFrac );

     }  // End of if ( fMipLevel > fThreshold - 1 )

     if ( bDisplayShadows == true )
     {
        float2 vLightRayTS = vLightTS.xy * fHeightMapRange;

        // Compute the soft blurry shadows taking into account
        // self-occlusion for features of the height field:

        float sh0 =  tex2Dgrad( tNormalMap, texSampleBase, dx, dy ).a;
        float shA = (tex2Dgrad( tNormalMap, texSampleBase + vLightRayTS
            * 0.88, dx, dy ).a - sh0 - 0.88 ) *  1 * fShadowSoftening;
        float sh9 = (tex2Dgrad( tNormalMap, texSampleBase + vLightRayTS *
             0.77, dx, dy ).a - sh0 - 0.77 ) *  2 * fShadowSoftening;
        float sh8 = (tex2Dgrad( tNormalMap, texSampleBase + vLightRayTS *
             0.66, dx, dy ).a - sh0 - 0.66 ) *  4 * fShadowSoftening;
        float sh7 = (tex2Dgrad( tNormalMap, texSampleBase + vLightRayTS *
             0.55, dx, dy ).a - sh0 - 0.55 ) *  6 * fShadowSoftening;
        float sh6 = (tex2Dgrad( tNormalMap, texSampleBase + vLightRayTS *
             0.44, dx, dy ).a - sh0 - 0.44 ) *  8 * fShadowSoftening;
        float sh5 = (tex2Dgrad( tNormalMap, texSampleBase + vLightRayTS *
             0.33, dx, dy ).a - sh0 - 0.33 ) * 10 * fShadowSoftening;
        float sh4 = (tex2Dgrad( tNormalMap, texSampleBase + vLightRayTS *
             0.22, dx, dy ).a - sh0 - 0.22 ) * 12 * fShadowSoftening;

        // Compute the actual shadow strength:
        fOcclusionShadow = 1 - max( max( max( max( max( max( shA, sh9 ),
             sh8 ), sh7 ), sh6 ), sh5 ), sh4 );

        // The previous computation overbrightens the image, let's adjust
        // for that:
        fOcclusionShadow = fOcclusionShadow * 0.6 + 0.4;

     }   // End of if ( bAddShadows )

   }   // End of if ( fMipLevel <= (float) nLODThreshold )

   // Compute resulting color for the pixel:
   cResultColor = ComputeIllumination( texSample, vLightTS,
                                       vViewTS, fOcclusionShadow );

   if ( bVisualizeLOD )
   {
      cResultColor *= cLODColoring;
   }

   // Visualize currently computed mip level, tinting the color blue
   // if we are in the region outside of the threshold level:
   if ( bVisualizeMipLevel )
   {
      cResultColor = fMipLevel.xxxx;
   }

   // If using HDR rendering, make sure to tonemap the result color
   // prior to outputting it. But since this example isn't doing that,
   // we just output the computed result color here:
   return cResultColor;


}   // End of float4 ps_main(..)
```

Listing 2. *Parallax occlusion mapping algorithm implementation. Pixel shader, DirectX 9.0c shader model 3.0*