Iterative Parallax Mapping with Slope Information

Mátyás Premecz[*]

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

Abstract

This paper improves per-pixel displacement mapping algorithms. First we review the previous work including bump, parallax, and relief mapping, and also sphere tracing. Then we propose a new iterative algorithm based on parallax mapping, which takes into account the surface slope represented by the stored normal vector. We show that the new method is much better than parallax mapping, however a little poorer than relief mapping, in terms of image quality. However, it is much faster than relief mapping, and is thus a good candidate for implementation in games.

Keywords: Displacement mapping, GPU, texture mapping

1 Introduction

Object geometry is usually defined on three scales, the macrostructure level, the mesostructure level, and the microstructure level. A geometric model usually refers to the macrostructure level, and is often specified as a set of polygonal or curved surfaces. The mesostructure level includes geometric details that are relatively small but still visible such as bumps on a surface. The microstructure level involves surface microfacets that are visually indistinguishable by human eyes, and are modeled by BRDFs [CT81] and conventional textures.

Displacement mapping [Coo84, CCC87] provides high geometric detail by adding mesostructure properties to the macrostructure model. This is done by modulating the smooth version of the model by a *height map* describing the difference between the simplified and the detailed model (figure 1).

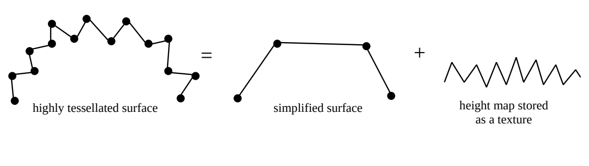


Figure 1: The basic idea of displacement mapping

[*]pmat@freemail.hu

Displacement mapping algorithms take sample points and displace them perpendicularly to the surface normal of the macrostructure surface with the distance obtained from the height map. The sample points can be either the vertices of the original or tessellated mesh (*per-vertex displacement mapping*) or the points corresponding to the texel centers (*per-pixel displacement mapping*). In case of per-vertex displacement mapping the modified geometry goes through the rendering pipeline. However, in per-pixel displacement mapping, the surface details are added in the last steps when texturing takes place. The idea of combining displacement mapping with texture lookups was proposed by Patterson, who called his method as *inverse displacement mapping* [PHL91]. On the GPU per-vertex displacement mapping can be implemented by the vertex shader. Per-pixel displacement mapping, on the other hand, is executed by the pixel shader.

During displacement mapping, the perturbed normal vectors should also be computed for illumination and self-shadowing information is also often needed.

In this paper we consider only pixel shader approaches. Since all of them work in *tangent space*, first the tangent space is revisited.

1.1 Tangent space

Tangent space is a coordinate system attached to the local surface [Gat03]. Its basis vectors are *normal* $\vec{N}$ that is perpendicular to the local surface, *tangent* $\vec{T}$ and *binormal* $\vec{B}$ vectors that are in the tangent plane of the surface and are perpendicular to the normal. The tangent vector points into the direction where the first texture coordinate u increases, while binormal points to the direction where the second texture coordinate v increases (figure 2).

Let us first consider the computation of the appropriate tangent and binormal vectors. Suppose that we have a triangle with vertices $\vec{p}_1, \vec{p}_2, \vec{p}_3$ in *local modeling space*, and with texture coordinates $[u_1, v_1]$, $[u_2, v_2]$, $[u_3, v_3]$. According to their definition, unknown tangent $\vec{T}$ and binormal $\vec{B}$ correspond to *texture space vectors* $[1, 0]$ and $[0, 1]$, respectively. The mapping between the texture space and the local modeling space is linear, thus an arbitrary point $\vec{p}$ is the following function of the texture coordinates:

$$\vec{p}(u, v) = \vec{p}_1 + (u - u_1) \cdot \vec{T} + (v - v_1) \cdot \vec{B}. \quad (1)$$

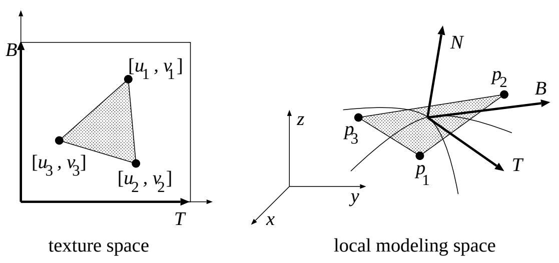


Figure 2: Tangent space

Note that the linear approximation means that 3D vector $\vec{p}_2 - \vec{p}_1$ corresponds to texture space vector $[u_2 - u_1, v_2 - v_1]$, and 3D vector $\vec{p}_3 - \vec{p}_1$ corresponds to texture space vector $[u_3 - u_1, v_3 - v_1]$. Thus we can write:

$$\begin{aligned} \vec{p}_2 - \vec{p}_1 &= [u_2 - u_1] \cdot \vec{T} + [v_2 - v_1] \cdot \vec{B}, \\ \vec{p}_3 - \vec{p}_1 &= [u_3 - u_1] \cdot \vec{T} + [v_3 - v_1] \cdot \vec{B}. \end{aligned}$$

This a linear system of equations for the unknown $\vec{T}$ and $\vec{B}$ vectors (in fact, there are three systems, one for each of the x, y, and z coordinates of the vectors). Solving these systems, we obtain:

$$\vec{T} = \frac{(u_3 - u_1) \cdot (\vec{p}_2 - \vec{p}_1) - (u_2 - u_1) \cdot (\vec{p}_3 - \vec{p}_1)}{(u_3 - u_1) \cdot (v_2 - v_1) - (u_2 - u_1) \cdot (v_3 - v_1)},$$

$$\vec{B} = \frac{(u_2 - u_1) \cdot (\vec{p}_3 - \vec{p}_1) - (u_3 - u_1) \cdot (\vec{p}_2 - \vec{p}_1)}{(u_3 - u_1) \cdot (v_2 - v_1) - (u_2 - u_1) \cdot (v_3 - v_1)}.$$

The normal vector of the triangle can be obtained as the cross product of these vectors since the cross product is surely perpendicular to both vectors:

$$\vec{N} = \vec{T} \times \vec{B}.$$

We often use unit length normal vector $\vec{N}^0 = \vec{N}/|\vec{N}|$ instead of $\vec{N}$. Then the surfaces is displaced by $\vec{N}^0 \cdot h$.

Note that in texture space $\vec{T}, \vec{B}, \vec{N}$ are orthonormal, that is, they are orthogonal and have unit length. However, these vectors are usually not orthonormal in modeling space (the transformation between the texture and modeling spaces is not angle and distance preserving). We should be aware that the lengths of the tangent and binormal vectors in modeling space are usually not 1, but express the expansion or shrinking of the texels as they are mapped onto the surface. On the other hand, while the normal is orthogonal to both the tangent and the binormal, the tangent and the binormal vectors are not necessarily orthogonal. Of course in special cases, such as when a rectangle, sphere, cylinder, etc. are parameterized in the usual way, the orthogonality of these vectors are preserved, but this is not true in the general case. Consider, for example, a sheared rectangle.

Having vectors $\vec{T}, \vec{B}, \vec{N}$ in the modeling space and point $\vec{p}_0$ corresponding to the origin of the texture space, a point (u, v, h) in tangent (i.e. texture) space can be transformed to modeling space as

$$\vec{p}(u, v) = \vec{p}_0 + [u, v, h] \cdot \begin{bmatrix} \vec{T} \\ \vec{B} \\ \vec{N} \end{bmatrix} = \vec{p}_0 + [u, v, h] \cdot \mathbf{M},$$

where $\mathbf{M}$ is the transformation matrix from tangent to modeling space. This matrix is also called sometimes as *TBN matrix*.

When transforming a vector $\vec{d} = \vec{p} - \vec{p}_0$ from modeling space to the tangent space, then the inverse of the matrix should be applied.

$$[u, v, h] = [d_x, d_y, d_z] \cdot \mathbf{M}^{-1}.$$

To compute the inverse, we can exploit only that the normal is orthogonal to the tangent and the binormal:

$$u = \frac{(\vec{T} \cdot \vec{d}) \cdot \vec{B}^2 - (\vec{B} \cdot \vec{d}) \cdot (\vec{B} \cdot \vec{T})}{\vec{B}^2 \cdot \vec{T}^2 - (\vec{B} \cdot \vec{T})^2},$$

$$v = \frac{(\vec{B} \cdot \vec{d}) \cdot \vec{T}^2 - (\vec{T} \cdot \vec{d}) \cdot (\vec{B} \cdot \vec{T})}{\vec{B}^2 \cdot \vec{T}^2 - (\vec{B} \cdot \vec{T})^2}, \quad h = \frac{\vec{N} \cdot \vec{d}}{\vec{N}^2}.$$

1.2 Storing height maps

When height function $h(u, v)$ is stored as a texture, we have to take into account that compact texture formats using a single byte per texel represent values in the range of $[0, 255]$ while the height function may be floating point and may have even negative values. Thus the stored values should be scaled and biased. Generally we use two constants `SCALE` and `BIAS` and convert the stored texel value $Texel(u, v)$ as:

$$h(u, v) = BIAS + SCALE \cdot Texel(u, v).$$

If we displace by $\vec{N}^0 \cdot h$, then constants `SCALE` and `BIAS` must be scaled when the object is scaled since we usually expect wrinkles also to grow together with the object.

1.3 Pixel shader implementation

Displacement mapping can be solved by the pixel shader (figure 3). The vertex shader processes only the smooth, simplified geometry, and we take into account the surface height map when fragments are processed, that is, at the end of the graphics pipeline. However, at this stage it is too late to change the geometry, thus the visibility problem needs to be solved in the pixel shader program by a ray-tracing like algorithm. The task can be imagined as tracing rays into the height field (figure 4).

The graphics pipeline processes the simplified geometry, and the pixel shader gets one of its points associated with texture coordinates $[u, v]$. This processed point has $(u, v, 0)$ coordinates in tangent space. The pixel shader

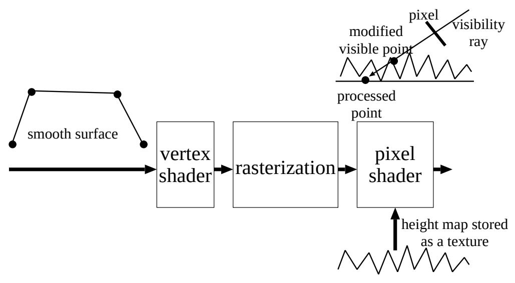


Figure 3: Displacement mapping on the pixel shader

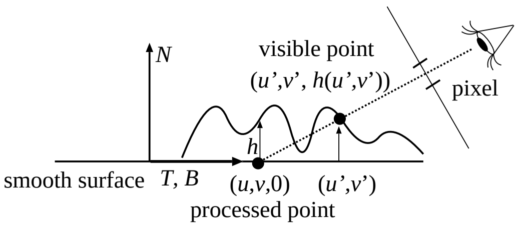


Figure 4: Ray tracing of the height field

program is responsible for finding that point of the height field which is really seen by the ray connecting the pixel center and processed point $(u, v, 0)$. This visible point has $(u', v', h(u', v'))$ coordinates in tangent space.

There are quite a few difficulties to implement this idea:

- A larger part of the height field texture might be searched, thus the process can be slow. To preserve speed, most of the implementations obtain the $[u', v']$ modified texture coordinates only approximately.
- There might be several points $(u', v', h(u', v'))$ of the height field that can be projected on the pixel of the processed point. In this case we need that point which is closest to the eye similarly to classical ray tracing. However, it makes the search process even more complex. Many algorithms simply ignore this fact and obtain one, approximate solution, which might be incorrect in this sense.
- The pixel shader is invoked only if its corresponding point of the simplified geometry is not back facing and also visible in case of early z-test. Thus it can happen that a height map point is ignored because its corresponding point of the simplified geometry is not processed by the pixel shader. The possibility of this error can be reduced if the simplified geometry encloses the detailed surface, that is, the height field values are *negative*, but back facing simplified polygons still pose problems.
- When the neighborhood of point $(u, v, 0)$ is searched, we should take into account that not only the height field, but also the underlying simplified geometry might change. In the pixel shader we do not have access to the mesh information, therefore simply assume that the simplified geometry is the plane of the currently processed triangle. Of course, this assumption fails at triangle edges, which prohibits the correct display of the *silhouette* of the detailed object. A simple solution of this problem is to discard fragments when the modified texture coordinate is out of the processed triangle. To cope with this problem in a more general case, *local curvature* information should also be supplied with the triangles to allow the higher order (e.g. quadratic) approximation of the smooth surface farther from the processed point. We implemented only the simple fragment discarding approach.
- Replacing processed point $(u, v, 0)$ by really visible point $(u', v', h(u', v'))$ does not change the pixel in which the point is visible, but modifies the depth value used to determine visibility in the z-buffer. Although it is possible to change this value in the pixel shader, we usually do not do that, because such change would have performance penalty due to the automatic disabling of the early z-culling. On the other hand, the height field modifies the geometry on a small scale, thus ignoring the z-changes before z-buffering usually does not create visible errors.

2 Previous work

2.1 Bump mapping

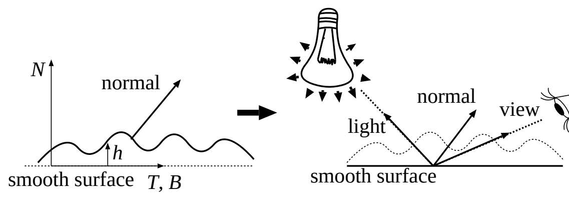


Figure 5: Bump mapping

Bump mapping [Bli78] can be seen as a strongly simplified version of displacement mapping. This technique was implemented in hardware even before the emergence of programmable GPUs [PAC97, Kil00, BERW97]. If the uneven surface has very small bumps it can be estimated as being totally flat. In case of flat surfaces, the approximated visible point $(u', v', h(u', v'))$ is equal to the processed point $(u, v, 0)$. However, in order to visualize bumps, it is necessary to simulate how light affects them. The simplest way is to calculate the perturbed normals for every pixel and use these normals to perform lighting.

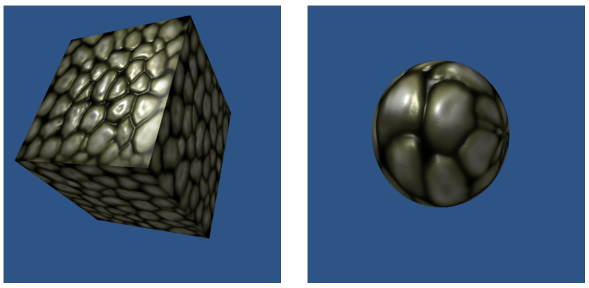


Figure 6: Bump mapping results at 500 FPS on an NV6800GT GPU.

2.2 Parallax mapping

Bump mapping controls only the shading normals according to the height field data, but does not distort the texture coordinates and the geometric data. *Parallax mapping* [KKI*01] still works with the simplified geometry, but also takes into account the height field when texture coordinates are obtained.

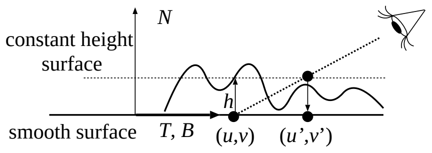


Figure 7: Parallax mapping

The texture coordinates used to index surface data textures (color, shininess, etc.) are modified according to the height of the actual point. As can be seen in figure 7, the original (u, v) texture coordinates get substituted by (u', v'), which are calculated from the direction of tangent-space view vector $\vec{V} = (V_x, V_y, V_z)$ and height value $h(u, v)$ read from a texture at point (u, v), assuming that the height field is constant $h(u, v)$ everywhere in the neighborhood of (u, v). Using similarity

$$(u' - u, v' - v) : h = (V_x, V_y) : V_z,$$

the modified texture coordinates are:

$$(u', v') = (u, v) + h(u, v) \cdot \left(\frac{V_x}{V_z}, \frac{V_y}{V_z}\right).$$

Parallax mapping is a computationally inexpensive, approximate 2D effect. Because of its ease of implementation, it can be used in conjunction with other types of mapping. The new texture coordinates can be used to index regular texture maps for coloring the surface, normal maps for bump mapping and computing reflection vectors, shininess maps for computing reflectivity, and probably any other standard type of texture data that the application may use. The modified texture coordinates are not perfect, so care must be taken in producing height maps and specifying scale and bias values. Because of the faulty assumption that height values are the same from one texture coordinate to the next, height maps with wildly varying heights tend to cause problems. Parallax mapping is incapable of detecting areas of a surface that occlude other areas, which is common on a surface with steep angles.

The implementation is very similar to the bump mapping pixel shader, the only difference is that the incoming texture coordinates are modified before indexing the normal and surface texture maps. The height values are in the alpha channel of the normal map.

```
float h = tex2D(hMap, uv).a * SCALE + BIAS;
return uv + h * View.xy / View.z;
```

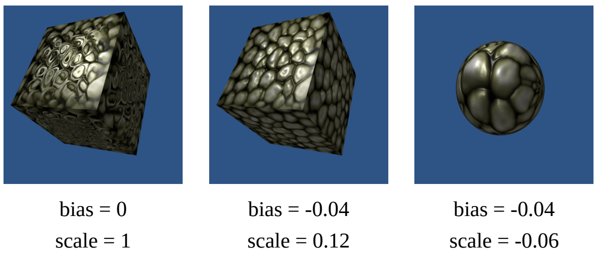


Figure 8: Parallax mapping results at 330 FPS. Note that this method is good if the scale is small and a smaller negative bias is added.

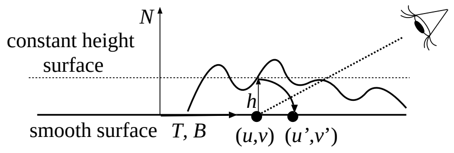


Figure 9: Parallax mapping with offset limiting

Parallax mapping in its original form has a significant flaw. The computation presented in the previous subsection assumes that the point at (u', v') has the same height as the point at (u, v), which is rarely true. Any surface that exhibits parallax will have varying heights. At nearly perpendicular viewing angles, texture coordinate offsets tend to be small. A small offset means that the height at (u', v') is likely to be very close to the height at (u, v). Consequently, the offset will be nearly correct. As the viewing angle becomes more grazing, offset values approach infinity. When offset values become significantly large, the odds of (u', v') indexing a similar height to that of (u, v) fade away, and the result seems to be random. This problem can reduce surfaces with complex height patterns to a shimmering mess of pixels that do not look anything like the original texture map. A simple solution to this problem is to limit the offsets so that they never grow longer than the original height in (u, v) [Wel04]. Examining original offset $h(u, v) \cdot (V_x/V_z, V_y/V_z)$, we can conclude that

- When the surface is seen at grazing angles and thus $V_z \ll V_x, V_y$, then offset limiting takes into effect, and the offset becomes $h(u, v) \cdot (V_x, V_y)$.

- When the surface is seen from a roughly perpendicular direction and thus $V_z \approx 1$, then the offset is again $h(u, v) \cdot (V_x, V_y)$ without any offset limiting.

Thus offset limiting can be implemented if the division by V_z is eliminated, which makes the implementation even simpler than that of the original parallax mapping. However, eliminating the division by V_z even when V_z is large causes the "swimming" of the texture, that is, the texture appears to slide over the surface.

Since parallax mapping is an approximation, any limiting value could be chosen, but this one works well enough and it reduces the code in the fragment program by two instructions. The implementation differs from the previous one only in the normalization of the view vector:

```
float h = tex2D(hMap, uv).a * SCALE + BIAS;
return uv + h * View.xy;
```

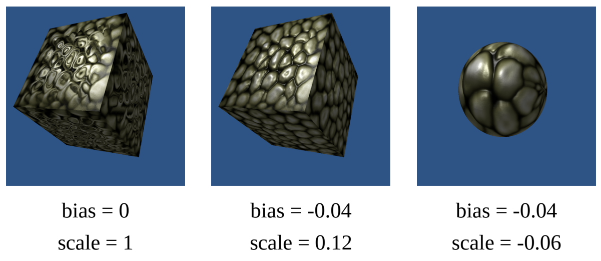


Figure 10: Parallax mapping with offset limiting results at 360 FPS.

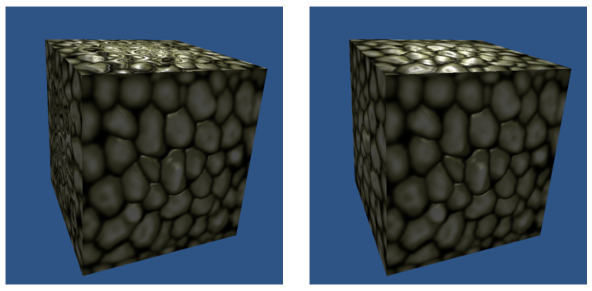


Figure 11: Comparison of parallax mapping methods without (left) and with (right) offset limiting. The bias is -0.12 and the scale is 0.18. Note that without offset limiting the image is bad when looking at the surface from grazing angles.

2.3 Relief mapping

Relief mapping [POC05, PO05] uses a root-finding approach. Just like the previous method, it is capable of

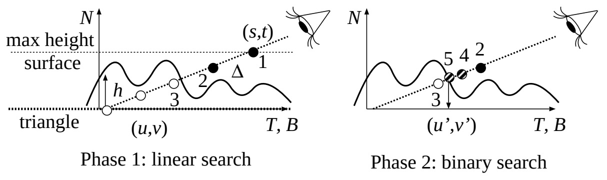


Figure 12: Relief mapping. The first phase of the algorithm is a linear search that stops at point 3. The second phase, the binary search, starts at point pair 2 and 3 and finds point 5 after two steps.

finding the first intersection, that is detecting occluded areas caused by steep angles on a surface. Thus, it gives a correct view of the detailed surface and handles self-shadowing, too. The key idea of relief mapping is an iterative approximate solution of the intersection calculation problem between a ray and an uneven surface given as a height field. One way to find this intersection point is to perform a binary search on the ray. The procedure has the following steps:

1. Calculate the view vector.

2. Transform the inverse view vector pointing from the viewer to the 3D position of the shaded point to tangent space ($-\vec{V}$).

3. The incoming texture coordinates (s, t) represent the point where the ray enters the area (highest possible level of surface).

4. With vector $-\vec{V}$ and (s, t) calculate (u, v), the point where the ray leaves this area (lowest possible level of surface).

5. Perform a texture lookup in the height field at the middle point of this segment.

6. If the given value is greater than the height of the middle point, continue with step 5 with the upper half of the original segment. Else continue with step 5 with the lower half of the original segment.

This procedure can be stopped after a given number of iterations or after reaching the needed precision.

```
float size = 1.0 / BIN_ITER;
float d = 1.0;  // depth
float bd = 0.0; // best depth

for (int i=0; i < BIN_ITER; i++) {
   size *= 0.5;
   float t= tex2D(hMap, dp + ds * (1-d)).a;
   if (d <= t) {
      bd = depth;
      d += 2 * size;
   }
   d -= size;
}
```

The binary search procedure just described may lead to incorrect results if the viewing ray intersects the height field surface in more than one point. In this case, the binary search might not find the first intersection point. In order to avoid this, we start the process with a linear search. Beginning at point (s, t), we step along the $(s, t) - (u, v)$ line at increments of $\Delta \cdot (s - u, t - v)$ looking for the first point inside the surface. Once the first point under the height field surface has been identified, the binary search starts using the last point outside the surface and current one. In this case, a smaller number of binary subdivisions is needed. For example, if the depth interval between two linearly searched points is $1/8$, a six-step binary search will be equivalent to subdividing the interval into 512 (8×2^6) equally spaced intervals.

The implementation is very similar to the parallax mapping pixel shader:

```
float dstep = 1.0 / LIN_ITER;   // depth step
float d = 1.0;                  //depth
float bd = 0.0; // best depth

// search from front to back
for ( int i=0; i < LIN_ITER; i++){
   d -= dstep;
   float h = tex2D(hMap, dp + ds * (1-d)).a;
   if (bd < 0.005) // if no depth found yet
      if (d <= h) bd = depth; // best depth
}
```

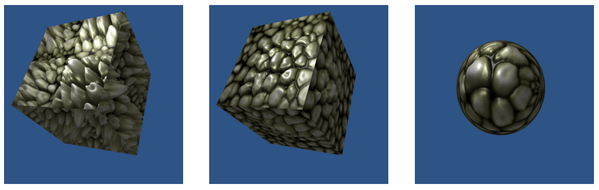


Figure 13: Relief mapping results at 40 FPS

Note that this implementation always make 8 linear search steps. If the graphics card supports shading model 3.0, the number of steps may vary from fragment to fragment as function of the angle between view direction and the interpolated surface normal at the fragment. According to experience, it is worth increasing the number of steps at oblique angles. This is possible as follows:

```
const int N = 8;
int LIN_ITER = lerp(2*N, N, View.z);
```

Note also that this algorithm always start at the maximum level surface, thus it is not affected by the `BIAS`.

This algorithm has been improved in several ways [BT04, YJ04, MM05]. For example, *steep Parallax Mapping* [MM05] emphasized the application of mip-mapping not to miss the first intersection, and exploited the dynamic looping of Shader Model 3.0.

2.4 Displacement mapping with sphere tracing

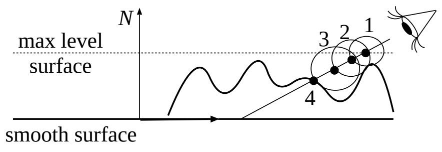


Figure 14: Displacement mapping with sphere tracing

Ray tracing of height fields is a numerical root finding algorithm, where we need the root that corresponds to the ray – height field intersection point closest to the eye. *Sphere tracing* uses a distance map and an iterative algorithm in order to always find the correct texture coordinates of a shaded pixel with correct self-occlusions of the displaced surface [Don05]. A distance map is a 3D texture that is precomputed based on the height field sample of the surface. The 3D texture is best to think of as an axis-aligned bounding box of the bumpy surface patch. Texels of the texture correspond to 3D points $\vec{p}$. The value stored in the texel is the distance from point $\vec{p}$ to the closest point on surface S. This distance can also be interpreted as the radius of the largest sphere of center $\vec{p}$, not intersecting, only touching the height field (figure 14). This property explains the name of the method. Formally, the 3D texture encodes the following distance function:

$$dist(\vec{p}, S) = \min\{|\vec{p}, \vec{q}| : \vec{q} \in S\}.$$

In order to find the first intersection of the ray and the height field, the ray is marched, that is, we make certain steps. When we are at point $\vec{p}$ on the ray, the distance field tells us that the surface is at least $dist(\vec{p}, S)$ far from this point. It means that we can step distance $dist(\vec{p}, S)$ safely on the ray, not risking that a hill of the height field is jumped over. The step gets closer to the first hit. Marching means the iteration of this step.

Formally, the iteration does the following. Suppose we have a ray with origin $\vec{p}_1$ (obtained as the intersection of the ray and the maximum level plane of the height field) and direction $-\vec{V}$ (normalized, tangent-space view vector pointing to the surface from the eye). We define a new point

$$\vec{p}_2 = \vec{p}_1 - dist(\vec{p}_1, S) \cdot \vec{V}.$$

Then the next point is $\vec{p}_3 = \vec{p}_2 - dist(\vec{p}_2, S) \cdot \vec{V}$, and so on. Each consecutive point is a little bit closer to the surface. Thus, if we take enough samples, our points converge to the closest intersection of the ray with the surface. The last point represents the displaced texture coordinates.

The implementation needs the Vertex Shader to compute the tangent-space view and light vectors. The distance map is fed to the pixel shader as a monochromatic, 3D texture.

```
float3 tex3D = float3(uv, 1);
for (int i = 0; i < ITERATION; i++) {
    float Dist = tex3D(distField, tex3D).a;
    tex3D -= Dist * View;
}
uv = tex3D.xy;
```

Note that this algorithm always start at the maximum level surface, thus it is not affected by the BIAS. On the other hand, the SCALE is burnt in the distance field, thus it cannot be controlled interactively.

The 3D distance field texture data may be generated by the *Danielsson's algorithm* [Dan80] implemented by the following program [Don05]. The distance field generation is quite time consuming. It is worth preparing it only once and storing the result in a file.

3 The new algorithm

All the methods proposed so far assume that the surface is locally horizontal, except for sphere tracing, which works with a locally spherical approximation. However, the height field and the normal map has more information, which has not been exploited so far. A better approximation can be obtained if we assume that the surface is still planar, but its normal vector can be arbitrary (i.e. this surface is not necessarily parallel with the smooth surface). The normal of the approximating plane can be taken as the normal vector read from the normal map, thus this approach does not require any further texture lookups.

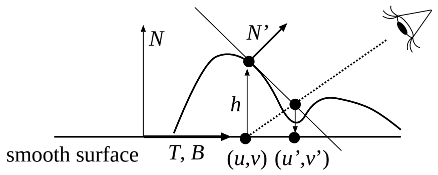


Figure 15: Parallax mapping taking into account the slope of the surface

A place vector of the approximating surface is $(u, v, h(u, v))$. The normal vector of this surface is the shading normal $\vec{N}'(u, v)$ read from the normal map at (u, v). The parametric equation of the view ray is $(u, v, 0) + \vec{V} \cdot t$.

Substituting the ray equation into the the equation of the approximating plane, we get

$$\vec{N}' \cdot ((u, v, 0) + \vec{V} \cdot t) = \vec{N}' \cdot (u, v, h).$$

Expressing ray parameter t and the point of intersection (u', v', h'), we obtain:

$$t = h \cdot \frac{N'_z}{(\vec{N}' \cdot \vec{V})}, \quad (u', v', h') = (u, v, 0) + h \cdot \frac{N'_z}{(\vec{N}' \cdot \vec{V})} \vec{V}.$$

We need only the first two coordinates as texel coordinate offsets:

$$(u', v') = (u, v) + h \cdot \frac{N'_z}{(\vec{N}' \cdot \vec{V})} (V_x, V_y).$$

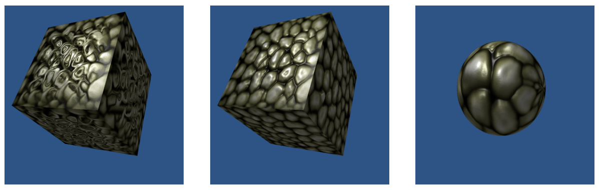
As we pointed out in the section on offset limiting, if $(\vec{N}' \cdot \vec{V})$ is small, the offset may be too big, so we should rather use a “safer” modification:

$$(u', v') \approx (u, v) + h \cdot N'_z \cdot (V_x, V_y).$$

Thus the shader calls the following function:

```
float4 Normal = tex2D(hMap, uv);
float h = Normal.a * SCALE + BIAS;
uv += h * Normal.z * View.xy;
```

This is as simple as the normal parallax mapping, but provides much better results.



bias = 0 | bias = -0.04 | bias = -0.04

scale = 1 | scale = 0.12 | scale = -0.06

Figure 16: Parallax mapping with slope information results at 355 FPS

Parallax mapping makes an attempt to offset the texture coordinates toward the really seen height field point. Of course, using a single attempt, no exact results can be expected. The accuracy of the solution, however, can be improved by iterating the same parallax mapping step by a few (say 3–4) times.

After an attempt we get an approximation of the intersection (u_i, v_i, h_i). Substituting this into the ray equation:

$$\vec{N}' \cdot ((u_i, v_i, h_i) + \vec{V} \cdot t) = \vec{N}' \cdot (u_{i+1}, v_{i+1}, h_{i+1}).$$

Solving it for the updated approximation, and ignoring the division with $(\vec{N}' \cdot \vec{V}')$:

$$(u_{i+1}, v_{i+1}, h_{i+1}) \approx (u_i, v_i, h_i) + (h(u_i, v_i) - h_i) \cdot N'_z \cdot \vec{V}.$$

The pixel shader of the iterative parallax mapping is similar to that of the parallax mapping with slope information with the following differences:

```
for(int i = 0; i < ITERATION; i++) {
    float4 Normal = tex2D(hMap, uv);
    float h = Normal.a * SCALE + BIAS;
    uv += (h - uv.z) * Normal.z * View;
}
```

Iterative parallax mapping is equivalent to numerical root finding, which tries to solve the ray equation. However, there is a problem. This method cannot guarantee that the found intersection point is the closest to the camera (not even the convergence is guaranteed).

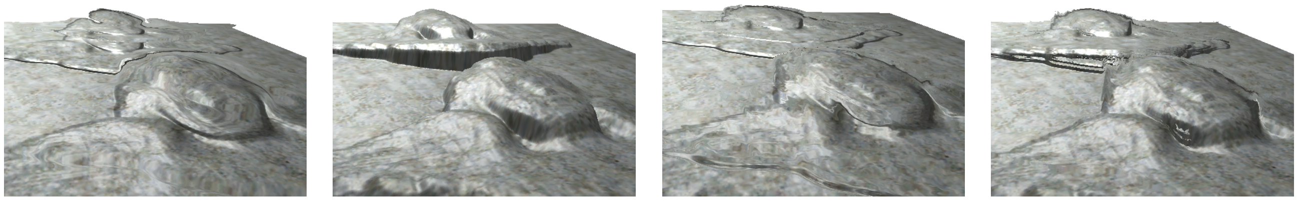


Figure 17: Comparison of the new methods to parallax and relief mapping using a simple silhouette processing

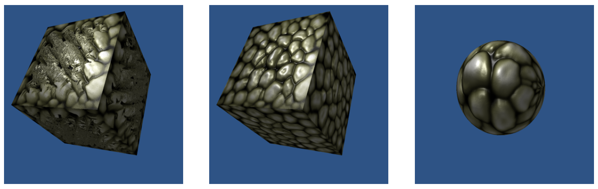


Figure 18: Iterative parallax mapping results at 246 FPS

4 Conclusions

We proposed a simple improvement of the parallax mapping algorithm, which is almost as fast as the original method but is close to the slower relief mapping in image quality.

5 Acknowledgement

This work has been supported by OTKA (ref. No.: T042735), GameTools FP6 (IST-2-004363) project, and by the National Office for Research and Technology (Hungary).

References

[BERW97] BENNEBROEK K., ERNST I., RÜSSELER H., WITTING O.: Design principles of hardware-based Phong shading and bump-mapping. *Computers and Graphics 21*, 2 (1997), 143–149.

[Bli78] BLINN J. F.: Simulation of wrinkled surfaces. In *Computer Graphics (SIGGRAPH '78 Proceedings)* (1978), pp. 286–292.

[BT04] BRAWLEY Z., TATARCHUK N.: Parallax occlusion mapping: Self-shadowing, perspective-correct bump mapping using reverse height map tracing. In *ShaderX3* (2004).

[CCC87] COOK R. L., CARPENTER L., CATMULL E.: The reyes image rendering architecture. In *Computer Graphics (SIGGRAPH '87 Proceedings)* (1987), pp. 95–102.

[Coo84] COOK R. L.: Shade trees. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1984), ACM Press, pp. 223–231.

[CT81] COOK R., TORRANCE K.: A reflectance model for computer graphics. *Computer Graphics 15*, 3 (1981).

[Dan80] DANIELSSON P.: Euclidean distance mapping. *Computer Graphics and Image Processing 14* (1980), 227–248.

[Don05] DONELLY W.: Per-pixel displacement mapping with distance functions. In *GPU Gems II*, M. P., (Ed.). Addison-Wesley, 2005, pp. 123–136.

[Gat03] GATH J.: Derivation of the tangent space matrix, 2003. http://www.blacksmith-studios.dk/projects/downloads/tangent_matrix_derivation.php.

[HDKS00] HEIDRICH W., DAUBERT K., KAUTZ J., SEIDEL H.-P.: Illuminating micro geometry based on precomputed visibility. In *SIGGRAPH 2000 Proceedings* (2000), pp. 455–464.

[Kil00] KILGARD M. J.: A practical and robust bump-mapping technique for today's GPUs. In *In GDC 2000: Advanced OpenGL Game Development* (2000).

[KKI*01] KANEKO T., KAKAHEI T., INAMI M., KAWAKAMI N., YANAGIDA Y., MAEDA T., TACHI S.: Detailed shape representation with parallax mapping. In *ICAT 2001* (2001).

[MM05] MCGUIRE M., MCGUIRE M.: Steep parallax mapping. In *I3D 2005 Poster* (2005). http://www.cs.brown.edu/research/graphics/games/SteepParallax/index.htm.

[PAC97] PEERCY M., AIREY J., CABRAL B.: Efficient bump mapping hardware. In *In SIGGRAPH 97 Conference Proceedings* (1997), pp. 303–306.

[PHL91] PATTERSON J. W., HOGGAR S. G., LOGIE J. R.: Inverse displacement mapping. *Computer Graphics Forum 10*, 2 (1991), 129–139.

[PO05] POLICARPO F., OLIVEIRA M. M.: Rendering surface details with relief mapping. In *ShaderX4: Advanced Rendering Techniques*, Engel W., (Ed.). Charles River Media, 2005.

[POC05] POLICARPO F., OLIVEIRA M. M., COMBA J.: Real-time relief mapping on arbitrary polygonal surfaces. In *ACM SIGGRAPH 2005 Symposium on Interactive 3D Graphics and Games* (2005), pp. 155–162.

[SKe95] SZIRMAY-KALOS (EDITOR) L.: *Theory of Three Dimensional Computer Graphics*. Akadémia Kiadó, Budapest, 1995. http://www.iit.bme.hu/˜szirmay.

[Wel04] WELSH T.: *Parallax Mapping with Offset Limiting: A Per-Pixel Approximation of Uneven Surfaces*. Tech. rep., Infiscape Corporation, 2004.

[YJ04] YEREX K., JAGERSAND M.: Displacement mapping with ray-casting in hardware. In *Siggraph 2004 Sketches* (2004).