Volume 25 (2006), Number 3 pp. 1–24

Displacement Mapping on the GPU — State of the Art

László Szirmay-Kalos, Tamás Umenhoffer

Department of Control Engineering and Information Technology, Budapest University of Technology, Hungary
Email: szirmay@iit.bme.hu

Abstract

This paper reviews the latest developments of displacement mapping algorithms implemented on the vertex, geometry, and fragment shaders of graphics cards. Displacement mapping algorithms are classified as per-vertex and per-pixel methods. Per-pixel approaches are further categorized as safe algorithms that aim at correct solutions in all cases, to unsafe techniques that may fail in extreme cases but are usually much faster than safe algorithms, and to combined methods that exploit the robustness of safe and the speed of unsafe techniques. We discuss the possible roles of vertex, geometry, and fragment shaders to implement these algorithms. Then the particular GPU based bump, parallax, relief, sphere, horizon mapping, cone stepping, local ray tracing, pyramidal and view-dependent displacement mapping methods, as well as their numerous variations are reviewed providing also implementation details of the shader programs. We present these methods using uniform notations and also point out when different authors called similar concepts differently. In addition to basic displacement mapping, self-shadowing and silhouette processing are also reviewed. Based on our experiences gained having re-implemented these methods, their performance and quality are compared, and the advantages and disadvantages are fairly presented.

Keywords: Displacement mapping, Tangent space, Direct3D 9 and 10, HLSL, Silhouettes, Self shadowing, GPU

ACM CCS: I.3.7 Three-Dimensional Graphics and Realism

1. Introduction

Object geometry is usually defined on three scales, the *macrostructure* level, the *mesostructure* level, and the *microstructure* level. A geometric model refers to the macrostructure level and is often specified as a set of polygonal surfaces. The mesostructure level includes higher frequency 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, HTSG91, APS00, KSK01] and conventional textures [BN76, Bli77, CG85, Hec86].

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

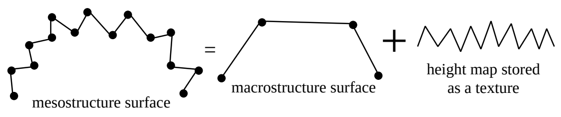


Figure 1: *The basic idea of displacement mapping*

Displacement mapping algorithms take *sample points* and displace them perpendicularly to the 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, surface details are added when color 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].

Inverse displacement mapping algorithms became popular also in CPU implementations. CPU based approaches [Tai92, LP95] flattened the base surface by warping and casted a curved ray, which was intersected with the displacement map as if it were a height field. More recent methods have explored direct ray tracing using techniques such as affine arithmetic [HS98], sophisticated caching schemes [PH96] and grid base intersections [SSS00]. Improvements of height field ray-tracing algorithms have also been proposed by [CORLS96, HS04, LS95, Mus88]. Huamin Qu et al. [QQZ*03] proposed a hybrid approach, which has the features of both rasterization and ray tracing.

On the GPU per-vertex displacement mapping can be implemented by the vertex shader or by the geometry shader. Per-pixel displacement mapping, on the other hand, is executed by the fragment shader. During displacement mapping, the perturbed normal vectors should also be computed for illumination, and self-shadowing information is also often needed.

In this review both vertex shader and fragment shader approaches are discussed and compared. Performance measurements have been made on NVidia GeForce 6800 GT graphics cards. When we briefly address geometry shader algorithms, an NVidia GeForce 8800 card is used for performance measurements. Shader code samples are in HLSL.

2. Theory of displacement mapping

Let us denote the mesostructure surface by the parametric form $\vec{r}(u,v)$, the macrostructure surface by $\vec{p}(u,v)$, the unit normal of the macrostructure surface by $\vec{N}^0(u,v)$, and the displacement by scalar function $h(u,v)$ called the *height map*. Vectors are defined by coordinates in 3D *modeling space*. Parameters u,v are in the unit interval, and are also called *texture coordinates*, while the 2D parameter space is often referred to as *texture space*. The height map is in fact a gray scale texture. Displacement mapping decomposes the definition of the surface to the macrostructure geometry and to a height map describing the difference of the mesostructure and macrostructure surfaces in the direction of the macrostructure normal vector:

$$\vec{r}(u,v) = \vec{p}(u,v) + \vec{N}^0(u,v)h(u,v). \tag{1}$$

Macrostructure surface $\vec{p}(u,v)$ is assumed to be a triangle mesh. Let us examine a single triangle of the mesh defined by vertices $\vec{p}_0, \vec{p}_1, \vec{p}_2$ in *modeling space*, which are associated with texture coordinates $[u_0,v_0],[u_1,v_1],[u_2,v_2]$, respectively. In order to find a parametric equation for the plane of the triangle, we select two basis vectors in the plane of the triangle, called *tangent* and *binormal*. One possibility is to define tangent vector $\vec{T}$ as the vector pointing into the direction where the first texture coordinate u increases, while binormal $\vec{B}$ is obtained as the vector pointing into the direction where the second texture coordinate v increases. It means that tangent $\vec{T}$ and binormal $\vec{B}$ correspond to *texture space vectors* $[1,0]$ and $[0,1]$, respectively. The plane of the triangle can be parameterized linearly, thus an arbitrary point $\vec{p}$ inside the triangle is the following function of the texture coordinates:

$$\vec{p}(u,v) = \vec{p}_0 + (u-u_0)\vec{T} + (v-v_0)\vec{B}. \tag{2}$$

We note that triangle meshes may also be regarded as the first-order (i.e. linear) approximation of the parametric equation of the mesostructure surface, $\vec{r}(u,v)$. Computing the first terms of its Taylor's series expansion, we get

$$\vec{r}(u,v) \approx \vec{r}(u_0,v_0) + (u-u_0)\frac{\partial \vec{r}}{\partial u} + (v-v_0)\frac{\partial \vec{r}}{\partial v}, \tag{3}$$

where the derivatives are evaluated at u_0,v_0. Comparing this equation to equation 2 we can conclude that

$$\vec{r}(u_0,v_0) = \vec{p}(u_0,v_0) = \vec{p}_0, \quad \vec{T} = \frac{\partial \vec{r}}{\partial u}, \quad \vec{B} = \frac{\partial \vec{r}}{\partial v}. \tag{4}$$

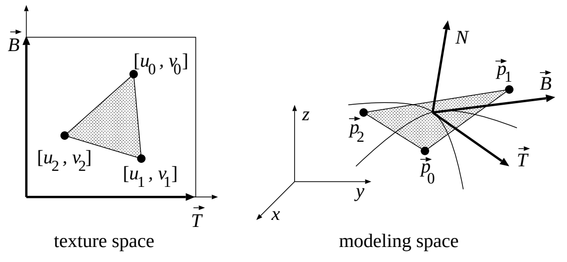


Figure 2: *Tangent space*

Tangent and binormal vectors together with the normal of the macrostructure triangle form a coordinate system that is attached to the macrostructure surface. This coordinate system defines the *tangent space* [Kil00, Gat03] (figure 2). We note that other definitions of the binormal are also possible, which are discussed together with the advantages and disadvantages at the end of this section.

If the parametric equation of the mesostructure surface is not known, then the texturing information of the triangles can be used [Gat03] to find the tangent and binormal vectors of a triangle. Substituting triangle vertices $\vec{p}_1$ and $\vec{p}_2$ with their texture coordinates into equation 2 we obtain

$$\begin{aligned}\vec{p}_1 - \vec{p}_0 &= (u_1-u_0)\vec{T} + (v_1-v_0)\vec{B},\\ \vec{p}_2 - \vec{p}_0 &= (u_2-u_0)\vec{T} + (v_2-v_0)\vec{B}.\end{aligned}$$

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

$$\vec{T} = \frac{(v_1-v_0)(\vec{p}_2-\vec{p}_0) - (v_2-v_0)(\vec{p}_1-\vec{p}_0)}{(u_2-u_0)(v_1-v_0) - (u_1-u_0)(v_2-v_0)},$$

$$\vec{B} = \frac{(u_1-u_0)(\vec{p}_2-\vec{p}_0)-(u_2-u_0)(\vec{p}_1-\vec{p}_0)}{(u_1-u_0)(v_2-v_0)-(u_2-u_0)(v_1-v_0)}. \quad (5)$$

The normal vector of the triangle can be obtained as the cross product of these vectors:

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

We often use the unit length normal vector $\vec{N}^0 = \vec{N}/|\vec{N}|$ instead of $\vec{N}$. Then the surface is displaced by $\vec{N}^0 h$. However, there is a problem. Suppose that the object is scaled uniformly by a factor s. In this case we expect bumps also to grow similarly, but $\vec{N}^0 h$ remains constant. Neither would the elimination of the normalization work, since if the displacement were defined as $\vec{N}h = (\vec{T}\times\vec{B})h$, then scaling by s would multiply the displacement distances by s^2. In order to avoid this, we can set the normal vector that is multiplied by the height value as [Bli78]

$$\vec{N} = \frac{\vec{T}\times\vec{B}}{\sqrt{(|\vec{T}|^2+|\vec{B}|^2)/2}}. \quad (6)$$

The selection between alternative $\vec{N}^0 h$ and equation 6 is more a modeling than a rendering issue. If we use unit normals, then the displacement is independent of the size of the triangle, and also of the scaling of the object. This makes the design of the bumps easy and the scaling problem can be solved by scaling the height function generally for the object. Using equation 6, on the other hand, the bumps will be higher on triangles being large in modeling space but being small in texture space. This makes the design of the height map more difficult. However, if we use a good parametrization, where the texture to model space transformation have roughly the same area expansion everywhere, then this problem is eliminated. In this second case, uniform scaling would not modify the appearence of the object, i.e. the relative sizes of the bumps.

In the remaining part of this paper we use the notation $\vec{N}$ to refer to the normal vector used for displacement calculation. This can be either the unit normal, or the normal scaled according to the length of the tangent and binormal vectors.

Note that in tangent space $\vec{T},\vec{B},\vec{N}$ are orthonormal, that is, they are orthogonal and have unit length. However, these vectors are not necessarily orthonormal in modeling space (the transformation between the texture and modeling spaces is not always 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, rotational surface, etc. are parameterized in the usual way, the orthogonality of these vectors is 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 tangent space, a point (u,v,h) in tangent 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 space to modeling space. This matrix is also called sometimes as *TBN matrix*.

When transforming a vector $\vec{d} = \vec{p} - \vec{p}_0$, for example the view and light vectors, from modeling space to 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, in the general case we can exploit only that the normal is orthogonal to the tangent and the binormal:

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

$$v = \frac{(\vec{B}\cdot\vec{d})\vec{T}^2 - (\vec{T}\cdot\vec{d})(\vec{B}\cdot\vec{T})}{\vec{B}^2\vec{T}^2-(\vec{B}\cdot\vec{T})^2}, \quad h = \frac{\vec{N}\cdot\vec{d}}{\vec{N}^2}. \quad (7)$$

If vectors $\vec{T},\vec{B},\vec{N}$ are orthogonal to each other, then these equations have simpler forms:

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

If vectors $\vec{T},\vec{B},\vec{N}$ were both orthogonal and had unit length, then the inverse of matrix $\mathbf{M}$ could be computed by simply transposing the matrix. This is an important advantage, so tangent and binormal vectors are also often defined according to this requirement. Having obtained vectors $\vec{T}$, $\vec{B}$ using either equation 4 or equation 5, and then $\vec{N}$ as their cross product, binormal $\vec{B}$ is recomputed as $\vec{B} = \vec{N}\times\vec{T}$, and finally all three vectors are normalized. The advantages of orthonormal $\vec{T}$, $\vec{B}$, $\vec{N}$ vectors are the easy transformation between tangent and modeling spaces, and the freedom of evaluating the illumination also in tangent space since the transformation to tangent space is *conformal* i.e. angle preserving. Evaluating the illumination in tangent space is faster than in world space since light and view vectors change smoothly so they can be transformed to tangent space per vertex, i.e. by the vertex shader, interpolated by the graphics hardware, and used the interpolated light and view vectors per fragment by the fragment shader. The disadvantage of the *orthonormalization* process is that we lose the intuitive interpretation that $\vec{T}$ and $\vec{B}$ show the effects of increasing texture coordinates u and v, respectively, and we cannot imagine 3D tangent space basis vectors as adding a third vector to the basis vectors of the 2D texture space.

When height function $h(u,v)$ is stored as a texture, we have to take into account that compact texture formats represent values in the range of $[0,1]$ with at most 8 bit fixed point precision, while the height function may have a higher range 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

$$h(u,v) = BIAS + SCALE \cdot Texel(u,v). \tag{8}$$

Height maps are often stored in the alpha channel of conventional color textures. Such representations are called *relief textures* [OB99, OBM00]. Relief textures and their extensions are also used in image based rendering algorithms [Oli00, EY03, PS02].

Although this review deals with those displacement mapping algorithms which store the height map in a two dimensional texture, we mention that three dimensional textures also received attention. Dietrich [Die00] introduced *elevation maps*, converting the height field to a texture volume. This method can lead to visual artifacts at grazing angles, where the viewer can see through the spaces between the slices. Kautz and Seidel [KS01] extended Dietrich's method and minimized the errors at grazing angles. Lengyel used a rendering technique to display fur interactively on arbitrary surfaces [LPFH01].

2.1. Lighting displacement mapped surfaces

When mesostructure geometry $\vec{r} = \vec{p} + \vec{N}^0 h$ is shaded, its real normal vector should be inserted into the illumination formulae. The mesostructure normal vector can be obtained in modeling space as the cross product of two vectors in its tangent plane. These vectors can be the derivatives according to texture coordinates u,v [Bli78]. Using equation 1 we obtain

$$\frac{\partial \vec{r}}{\partial u} = \frac{\partial \vec{p}}{\partial u} + \vec{N}^0 \frac{\partial h}{\partial u} + \frac{\partial \vec{N}^0}{\partial u} h \approx \vec{T} + \vec{N}^0 \frac{\partial h}{\partial u}$$

since $\partial \vec{p}/\partial u = \vec{T}$ and on a smooth macrostructure surface $\partial \vec{N}^0/\partial u \approx 0$. Similarly, the partial derivative according to v is

$$\frac{\partial \vec{r}}{\partial v} = \frac{\partial \vec{p}}{\partial v} + \vec{N}^0 \frac{\partial h}{\partial v} + \frac{\partial \vec{N}^0}{\partial v} h \approx \vec{B} + \vec{N}^0 \frac{\partial h}{\partial v}.$$

The mesostructure normal vector is the cross product of these derivatives:

$$\vec{N}' = \frac{\partial \vec{r}}{\partial u} \times \frac{\partial \vec{r}}{\partial v} = \vec{N} + (\vec{N}^0 \times \vec{B}) \frac{\partial h}{\partial u} + (\vec{T} \times \vec{N}^0) \frac{\partial h}{\partial v},$$

since $\vec{T} \times \vec{B} = \vec{N}$ and $\vec{N}^0 \times \vec{N}^0 = 0$.

In order to speed up the evaluation of the mesostructure normal, vectors $\vec{t} = \vec{N}^0 \times \vec{B}$ and $\vec{b} = \vec{T} \times \vec{N}^0$ can be precomputed on the CPU and passed to the vertex shader if illumination is computed per vertex, or to the fragment shader if illumination is computed per fragment. Since vectors $\vec{t}$ and $\vec{b}$ are in the tangent plane they can also play the roles of tangent and binormal vectors. Using these vectors the mesostructure normal is

$$\vec{N}' = \vec{N} + \vec{t} \frac{\partial h}{\partial u} + \vec{b} \frac{\partial h}{\partial v}. \tag{9}$$

The following fragment shader code computes the mesostructure normal according to this formula, replacing the derivatives by finite differences. The macrostructure normal $\vec{N}$, and tangent plane vectors $\vec{t}$, $\vec{b}$ are passed in registers and are denoted by `N`, `t`, and `b`, respectively. The texture coordinates of the current point are in `uv` and the displacement is in the alpha channel of texture map `hMap` of resolution `WIDTH`×`HEIGHT`.

```
float2 du=float2(1/WIDTH, 0);
float2 dv=float2(0, 1/HEIGHT);
float dhdu = SCALE/(2/WIDTH) *
                 (tex2D(hMap, uv+du).a -
                  tex2D(hMap, uv-du).a);
float dhdv = SCALE/(2/HEIGHT) *
                 (tex2D(hMap, uv+dv).a -
                  tex2D(hMap, uv-dv).a);
    // get model space normal vector
float3 mNormal = normalize(N+t*dhdu+b*dhdv);
```

On the other hand, instead of evaluating this formula to obtain mesostructure normals we can also use *normal maps* which store the mesostructure normals in textures. Height values and normal vectors are usually organized in a way that the r,g,b channels of a texel represent either the tangent space or the modeling space normal vector, and the alpha channel the height value.

Storing modeling space normal vectors in normal maps has the disadvantage that the normal map cannot be tiled onto curved surfaces since multiple tiles would associate the same texel with several points on the surface, which do not necessarily have the same mesostructure normal. Note that by storing tangent space normal vectors this problem is solved since in this case the mesostructure normal depends not only on the stored normal but also on the transformation between tangent and modeling spaces, which can follow the orientation change of curved faces.

Having the mesostructure normal, the next crucial problem is to select the coordinate system where we evaluate the illumination formula, for example, the Phong-Blinn reflection model [Bli77]. Light and view vectors are available in world or in camera space, while the shading normal is usually available in tangent space. The generally correct solution is to transform the normal vector to world or camera space and evaluate the illumination there. However, if the mappings between world space and modeling space, and between modeling space and tangent space are angle preserving, then view and light vectors can also be transformed to tangent space and we can compute the illumination formula here.

2.2. Obtaining height and normal maps

Height fields are natural representations in a variety of contexts, including e.g. water surface and terrain modeling. In these cases, height maps are provided by simulation or measurement processes. Height maps are gray scale images and can thus also be generated by 2D drawing tools. They can be the results of surface simplification when the difference of the detailed and the macrostructure surface is computed [CMSR98, Bla92, ATI03]. In this way height and normal map construction is closely related to *tessellation* [GH99, DH00, DKS01, MM02, EBAB05] and subdivision algorithms [Cat74, BS05]. Using light measurement tools and assuming that the surface is diffuse, the mesostructure of the surface can also be determined from reflection patterns using *photometric stereo* also called *shape from shading* techniques [ZTCS99, RTG97, LKG*03].

Displacement maps can also be the results of rendering during impostor generation, when complex objects are rasterized to textures, which are then displayed instead of the original models [JWP05]. Copying not only the color channels but also the depth buffer, the texture can be equipped with displacement values [OBM00, MJW07].

The height map texture is a discrete representation of a continuous function, thus can cause aliasing and sampling artifacts. Fournier [Fou92] pre-filtered height maps to avoid aliasing problems. Standard bi- and tri-linear interpolation of normal maps work well if the normal field is continuous, but may result in visible artifacts in the areas where the field is discontinuous, which is common for surfaces with creases and dents. Sophisticated filtering techniques based on *feature-based textures* [WMF*00, TC05, PRZ05] and *silhouette maps* [Sen04] have been proposed in the more general context of texture mapping to overcome these problems.

3. Per-vertex displacement mapping on the GPU

Displacement mapping can be implemented either in the vertex shader modifying the vertices, or in the fragment shader re-evaluating the visibility or modifying the texture coordinates. This section presents the vertex shader solution.

Graphics hardware up to Shader Model 3 (or Direct3D 9) is unable to change the topology of the triangle mesh, thus only the original vertices can be perturbed. This has been changed in Shader Model 4 (i.e. Direct3D 10) compatible hardware. Assuming Shader Model 3 GPUs the real modification of the surface geometry requires a highly tessellated, but smooth surface, which can be modulated by a height map in the vertex shader program.

3.1. Vertex modification on the vertex shader

The vertices of the triangles are displaced in the direction of the normal of the macrostructure surface according to the height map. New, mesostructure normal vectors are also assigned to the displaced vertices to accurately simulate the surface lighting. Since the introduction of Shader Model 3.0 compatible hardware, the vertex shader is allowed to access the texture memory, thus the height map can be stored in a texture. In earlier, Shader Model 1 or 2 compatible hardware only procedural displacement mapping could be executed in the vertex shader.

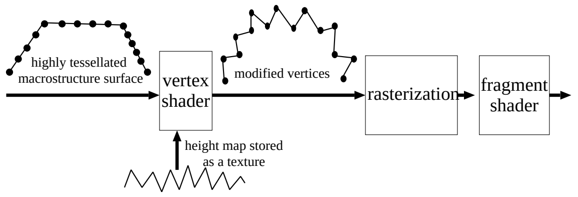

highly tessellated macrostructure surface
vertex shader
modified vertices
rasterization
fragment shader
height map stored as a texture

Figure 3: *Displacement mapping on the vertex shader*

The following vertex shader program takes modeling space point `Position` with macrostructure normal vector `Normal` and texture coordinates `uv`, reads height map `hMap`, computes modeling space vertex position `mPos`, and transforms the modified point to clipping space `hPos`:

```
float h = tex2Dlod(hMap, uv).a * SCALE + BIAS;
float3 mPos = Position + Normal * h;
hPos = mul(float4(mPos,1), WorldViewProj);
```

Figure 4 has been rendered by this vertex shader. Note that the macrostructure geometries, which are a square and a cylinder, should be finely tessellated to allow vertex displacement.

Per vertex displacement mapping really changes the geometry, thus it can handle cases when the surface is curved, and can provide correct *silhouettes* automatically, which is a great advantage.

The problems of vertex displacement methods are as follows:

- The number of the used vertices can be very high, which contradicts to that the aim of displacement mapping in hardware accelerated environment is to reduce the vertex number without losing surface detail.
- If the displacement is done on the GPU, performing shadow computations on CPU gives either incorrect results, or takes too long because the transformed data must be fed back to the CPU. Thus, the only way to go is to compute shadows on the GPU, which is rather problematic if shadow volumes are used [HLHS03] and the GPU does not have a geometry shader. Thus under Shader Model 4 we are better off if the depth mapped shadow method is implemented [Cro77, WSP04].

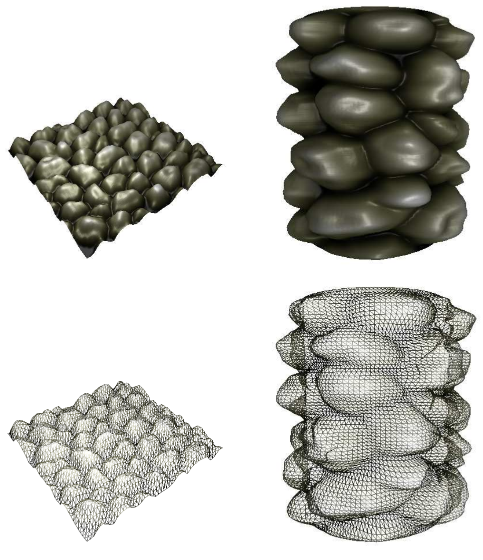


Figure 4: *Vertex displacement with shading (top row). The geometry should be finely tesselated (bottom row).*

- GPUs have usually more pixel-processing power than vertex-processing power, but the *unified shader architecture* [Sco07] can find a good balance.
- Pixel shaders are better equipped to access textures. Older GPUs do not allow texture access within a vertex shader. More recent GPUs do, but the access modes are limited, and the texture access in the vertex shader is slower than in the fragment shader.
- The vertex shader always executes once for each vertex in the model, but the fragment shader executes only once per pixel on the screen. This means that in fragment shaders the work is concentrated on nearby objects where it is needed the most, but vertex shader solutions devote the same effort to all parts of the geometry, even to invisible or hardly visible parts.

3.2. Shader Model 4 outlook

Shader Model 4 has introduced a new stage in the rendering pipeline between the vertex shader and the rasterizer unit, called the *geometry shader* [Bly06]. The geometry shader processes primitives and can create new vertices, thus it seems to be an ideal tessellator. So it becomes possible to implement displacement mapping in a way that the vertex shader transforms the macrostructure surface, the geometry shader tessellates and modulates it with the height map to generate the mesostructure mesh, which is output to the rasterizer unit.

Though subdividing the meshes with the geometry shader looks a good choice, care should be taken to implement this idea. The number of new triangles generated by the geometry shader is limited by the maximum output size, which is currently $1024 \times 32 = 32768$ bits. This limitation can be overcome if the data is fed back to the geometry shader again creating levels of subdivisions. One should take into account that subdividing a triangle into hundreds of triangles may give good quality results but highly reduces performance. The number of new triangles should depend on the frequency of the height map, and more vertices should be inserted into areas where the height map has high variation and less detailed tessellation is needed where the height map changes smoothly. The texture and model space areas of the triangles should also influence the number of subdivisions. The subdivision algorithm should also consider the orientation of the triangles according to the viewer as triangles perpendicular to the view ray may not require as many subdivisions, while faces seen at grazing angles, and especially silhouette edges should be subdivided into more pieces.

Developing geometry shader programs that meet all the requirements described above is not easy, and it will be an important research area in the near future.

4. Per-pixel displacement mapping on the GPU

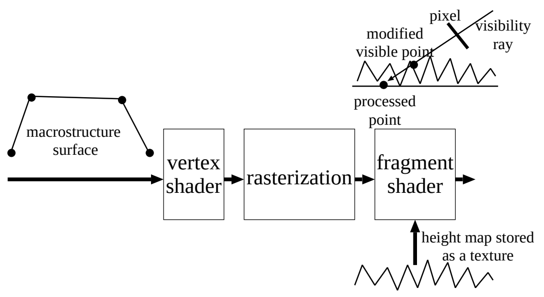


Figure 5: *Displacement mapping on the fragment shader*

Displacement mapping can also be solved by the fragment shader (figure 5). The vertex shader transforms only the macrostructure geometry, and the surface height map is taken into account when fragments are processed, that is, when color texturing takes place. However, at this stage it is too late to change the geometry, thus the visibility problem needs to be solved in the fragment shader program by a ray-tracing like algorithm. The task can be imagined as tracing rays into the height field (figure 6) to obtain the texture coordinates of the visible point, which are used to fetch color and normal vector data.

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

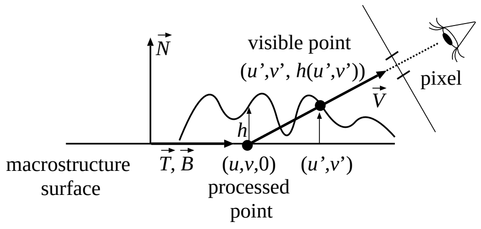


Figure 6: *Ray tracing of the height field*

program should find that point of the height field which is really seen by the ray connecting the pixel center and processed point $(u,v,0)$. The direction of this ray is defined by tangent space view vector $\vec{V}$. The *visible point* is on the height field and thus has tangent space coordinates $(u',v',h(u',v'))$ for some unknown (u',v'). This visible point is also on the ray of equation

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

for some ray parameter t, thus we need to solve the following equation:

$$(u',v',h(u',v')) = (u,v,0) + \vec{V}t \qquad (10)$$

for unknown u',v',t parameters.

Height function h is evaluated by fetching the texture memory and using equation 8 to scale and bias the stored value. If during the solution of ray equation 10 we should read and scale texels many times, it is more efficient to suppose that the height field values are in the range of $[0,1]$ and transform the view vector, i.e. the direction vector of the ray, appropriately:

$$V'_z = (V_z - BIAS)/SCALE.$$

The space after this scaling is called *normalized tangent space*. This transformation is often interpreted in the following way. Let us define the ray by the processed point (u,v) and by another tangent space point where the ray intersects the maximum height plane. This point may be called *entry point* and is given as coordinates $(u_{in},v_{in},1)$ in normalized tangent space since the ray enters here the volume of possible height field intersections. Similarly the tangent space processed point, $(u,v,0)$, on the other hand, can be considered as the *exit point*. With the entry and exit points the ray segment between the minimum and maximum height planes is

$$(u,v,0)(1-H) + (u_{in},v_{in},1)H, \quad H \in [0,1]. \qquad (11)$$

In this representation height value H directly plays the role of the ray parameter. The equation to be solved is

$$(u',v') = (u,v)(1-H) + (u_{in},v_{in})H, \quad h(u',v') = H.$$

There are quite a few difficulties to implement per-pixel displacement mapping:

- 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 solutions of the equation, i.e. several points $(u',v',h(u',v'))$ of the height field that can be projected onto the same pixel. Note that in figure 6 we can identify three such intersections. In this case we need that point which is the closest to the eye, i.e. has maximum t or H. We shall call this intersection as the "*first intersection*" of the ray. However, looking for the "first" intersection makes the search process even more complex. Many algorithms simply ignore this fact and obtain a solution that might be incorrect due to occlusions.
- The new, approximate texture coordinates (u',v') might fall outside the texture footprint of the given mesh, thus invalid texels or texels belonging to other meshes might be fetched. A possible solution is to separate textures of different meshes by a few texel wide boundaries, fill these boundaries by special "invalid" values, and discard the fragment if such invalid texel is fetched.
- The fragment 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 fragment 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 fragment shader we do not have access to the mesh information, therefore we 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. The *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.
- Replacing processed point $(u,v,0)$ by the really visible point defined by $(u',v',h(u',v'))$ 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 fragment shader, algorithms not always 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.

Per-pixel displacement mapping approaches can be mainly categorized according to how they attack the first two challenges of this list, that is, whether or not they aim at accurate intersection calculation and at always finding the first intersection.

Non-iterative or *single-step* methods correct the texture coordinates using just local information, thus they need only at most one extra texture access but may provide bad results.

Iterative methods explore the height field globally thus they can potentially find the exact intersection point but they read the texture memory many times for each fragment. Iterative techniques can be further classified to *safe methods* that try to guarantee that the first intersection is found, and to *unsafe methods* that may result in the second, third, etc. intersection point if the ray intersects the height field many times. Since unsafe methods are much faster than safe methods it makes sense to combine the two approaches. In *combined methods* first a safe iterative method reduces the search space to an interval where only one intersection exists, then an unsafe method computes the accurate intersection quickly.

In the following subsections we review different fragment shader implementations of the height field rendering.

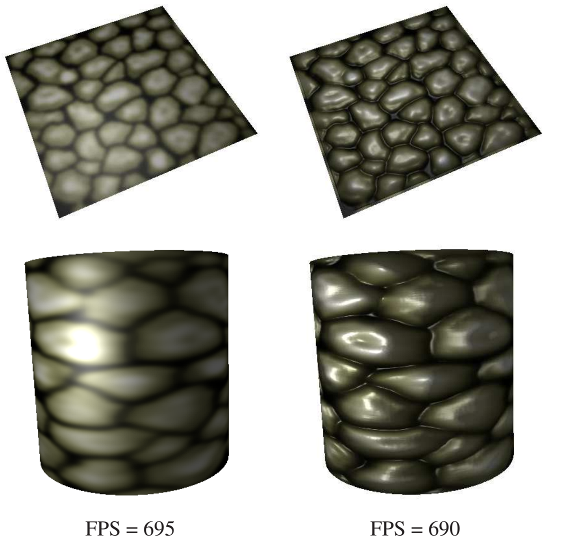


Figure 8: *Texture mapping (left) and texture mapping with bump mapping (right).*

4.1. Non-iterative methods

Non-iterative methods read the height map at the processed point and using this local information obtain more accurate texture coordinates, which will address the normal map and color textures.

4.1.1. Bump mapping

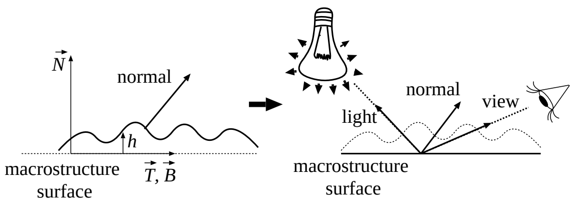


Figure 7: *Bump mapping*

Bump mapping [Bli78] can be seen as a strongly simplified version of displacement mapping (figure 7). We included bump mapping into this survey for the sake of completeness and to allow comparisons. 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 read the mesostructure normal vectors from a normal map and use these normals to perform lighting.

This technique was implemented in hardware even before the emergence of programmable GPUs [PAC97, Kil00, BERW97, TCRS00]. A particularly popular simplification used a simple *embossing* trick to simulate bump mapping for diffuse surfaces [Bli78, Sch94]. Becker and Max extended bump mapping to account for occlusions [BM93], and called their technique *redistribution bump-mapping*. They also described how to switch between three rendering techniques (BRDF, redistribution bump-mapping and displacement mapping) within a single object according to the amount of visible surface detail.

4.1.2. Parallax mapping

Taking into account the height at the processed point *parallax mapping* [KKI*01] not only controls the shading normals as bump mapping, but also modifies the texture coordinates used to obtain mesostructure normals and color data.

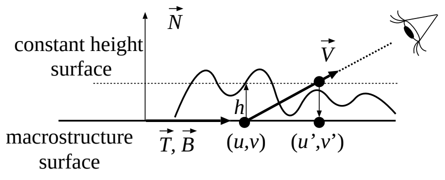


Figure 9: *Parallax mapping*

The texture coordinates are modified assuming that the height field is constant $h(u, v)$ everywhere in the neighborhood of (u, v). As can be seen in figure 9, 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). The assumption on a constant height surface simplifies the ray equation to

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

which has the following solution:

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

In the following implementation the tangent space view vector is denoted by `View`, and texture coordinates `uv` of the processed point are modified to provide the texture coordinates of the approximate visible point:

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

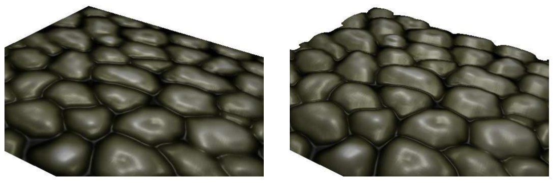


Figure 10: *Comparison of bump mapping (left) and parallax mapping (right) setting BIAS* $= -0.06$ *and SCALE* $= 0.08$.

Figure 10 compares bump mapping and parallax mapping. Note that at almost the same speed, parallax mapping provides more plausible bumps. However, parallax mapping in its original form has a significant flaw. As the viewing angle becomes more grazing, offset values approach infinity. When offset values become 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.

4.1.3. Parallax mapping with offset limiting

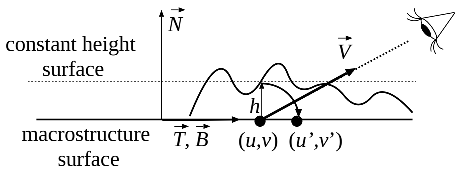


Figure 11: *Parallax mapping with offset limiting*

A simple way to solve the problem of parallax mapping at grazing angles is to limit the offsets so that they never get larger than the height at (u,v) [Wel04]. Asssuming the view vector to be normalized, and examining parallax offset $h(u,v)(V_x/V_z, V_y/V_z)$, 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 $h(u,v)(V_x,V_y)$.
- When the surface is seen from a roughly perpendicular direction and thus $V_z \approx 1$, then the offset is again $h(u,v)(V_x,V_y)$ 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 of offset limiting is as follows:

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

Note that offset limiting requires that the tangent space view is normalized. Since view vector normalization is usually needed for illumination as well, this requirement has no additional cost. Figure 12 demonstrates that offset limiting can indeed reduce the errors at grazing angles.

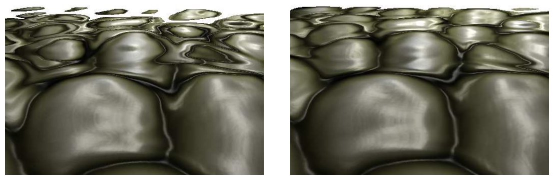


Figure 12: *Comparison of parallax mapping (left) and parallax mapping with offset limiting (right) setting BIAS* $= -0.14$ *and SCALE* $= 0.16$.

4.1.4. Parallax mapping taking into account the slope

Parallax mapping assumes that the surface is a constant height plane. 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 macrostructure 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 [MM05].

A place vector of the approximating plane is $(u,v,h(u,v))$. The normal vector of this plane is the shading normal $\vec{N}'(u,v)$ read from the normal map at (u,v). Substituting the ray equation into the equation of the approximating plane, we get

$$\vec{N}' \cdot ((u,v,0) + \vec{V}t) = \vec{N}' \cdot (u,v,h). \tag{12}$$

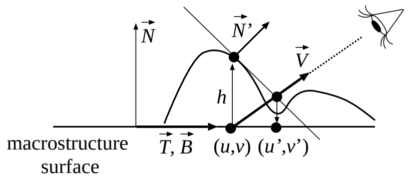


Figure 13: *Parallax mapping taking into account the slope*

Expressing ray parameter t and then the (u', v') coordinates of the intersection, we obtain

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

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:

$$(u', v') \approx (u, v) + hN'_z(V_x, V_y).$$

The fragment shader implementation is as follows:

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

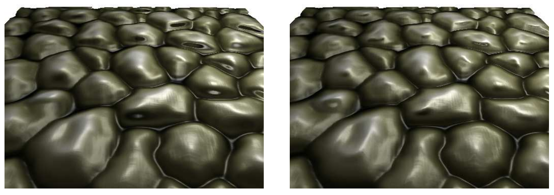


Figure 14: *Comparison of parallax mapping with offset limiting (left) and parallax mapping with slope information (right) using BIAS* $= -0.04$ *and SCALE* $= 0.12$.

This is as simple as the original parallax mapping, but provides much better results (figure 14).

4.2. Unsafe iterative methods

Iterative methods explore the height field globally to find the intersection between the ray and the height field. In the following subsections we review unsafe methods that obtain an intersection but not necessarily the first one, i.e. the one that is the closest to the eye.

4.2.1. Iterative parallax mapping

Parallax mapping makes an attempt to offset the texture coordinates toward the really seen height field point. Of course, with a single attempt perfect results cannot be expected. The accuracy of the solution, however, can be improved by repeating the correction step by a few (say 3–4) times [Pre06].

After an attempt we get an approximation of the intersection (u_i, v_i, H_i) that is on the ray. Substituting this into the ray equation, we get

$$\vec{N}' \cdot ((u_i, v_i, H_i) + \vec{V}\Delta t) = \vec{N}' \cdot (u_i, v_i, h(u_i, v_i)).$$

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

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

The fragment shader of the iterative parallax mapping is similar to that of the parallax mapping with slope information, but here we use a three dimensional tangent space point `uvh` that contains not only the texture coordinates but also the current height on the ray:

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

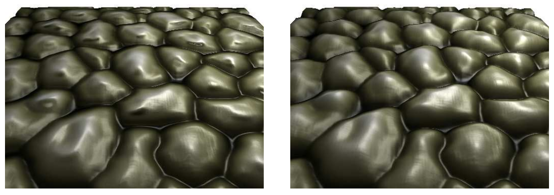


Figure 15: *Comparison of parallax mapping with slope (left) and iterative parallax mapping (right) setting BIAS* $= -0.04$, *SCALE* $= 0.12$, *and* `PAR_ITER`$= 2$.

Iterative parallax mapping is a fast but unsafe method. This method cannot guarantee that the found intersection point is the closest to the camera (in extreme cases not even the convergence is guaranteed). However, in many practical cases it is still worth using since it can cheaply but significantly improve parallax mapping (figure 15).

4.2.2. Binary search

Suppose we have two guesses on the ray that enclose the real intersection point since one guess is above while the other is below the height field. Considering the ray equation of form $(u, v, 0)(1 - H) + (u_{in}, v_{in}, 1)H$, points on the minimum and maximum height values, i.e. points defined by height parameters $H_{\min} = 0$ and $H_{\max} = 1$, surely meet this requirement.

Binary search halves the interval $(H_{\min}, H_{\max})$ containing the intersection in each iteration step putting the next guess at the middle of the current interval [POC05, PO05]. Comparing the height of this guess and the height field, we can

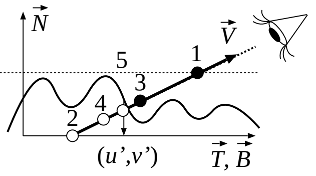


Figure 16: *Binary search that stops at point 5*

decide whether or not the middle point is below the surface. Then we keep that half interval where one endpoint is above while the other is below the height field.

In the following implementation the texture coordinates at the entry and exit points are denoted by `uvin` and `uvout`, respectively, and the height values at the two endpoints of the current interval are `Hmin` and `Hmax`.

```
float2 uv; // texture coords of intersection
for (int i = 0; i < BIN_ITER; i++) {
   H = (Hmin + Hmax)/2;    // middle
   uv = uvin * H + uvout * (1-H);
   float h = tex2D(hMap, uv).a;
   if (H <= h) Hmin = H; // below
   else        Hmax = H; // above
}
```

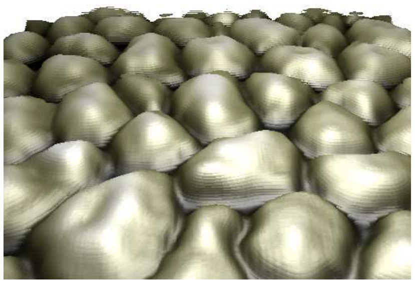


Figure 17: *Binary search using 5 iteration steps. The rendering speed is 455 FPS.*

The binary search procedure quickly converges to an intersection but may not result in the first intersection that has the maximum height value (figure 17).

4.2.3. Secant method

Binary search simply halves the interval potentially containing the intersection point without taking into account the underlying geometry. The *secant method* [YJ04, RSP06] on the other hand, assumes that the surface is planar between the two guesses and computes the intersection between the planar surface and the ray. It means that if the surface were really a plane between the first two candidate points, this intersection point could be obtained at once. The height field is checked at the intersection point and one endpoint of the current interval is replaced keeping always a pair of end points where one end is above while the other is below the height field. As has been pointed out in [SKALP05] the name "secant" is not precise from mathematical point of view since the secant method in mathematics always keeps the last two guesses and does not care of having a pair of points that enclose the intersection. Mathematically the root finding scheme used in displacement mapping is equivalent to the *false position method*. We note that it would be worth checking the real secant algorithm as well, which almost always converges faster than the false position method, but unfortunately, its convergence is not guaranteed in all cases.

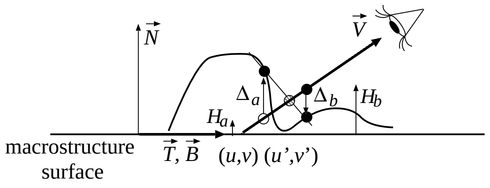


Figure 18: *The secant method*

To discuss the implementation of the secant method, let us consider the last two points of the linear search, point $\vec{b}$ that is below the height field, and $\vec{a}$ that is the last point which is above the height field. Points $\vec{b}$ and $\vec{a}$ are associated with height parameters H_b and H_a, respectively. The level differences between the points on the ray and the height field at $\vec{a}$ and $\vec{b}$ are denoted by Δ_a and Δ_b, respectively (note that $\Delta_a \geq 0$ and $\Delta_b < 0$ hold). Figure 18 depicts these points in the plane containing viewing vector $\vec{V}$. We wish to obtain the intersection point of the view ray and of the line between the points on the height field just above $\vec{a}$ and below $\vec{b}$.

Using figure 18 and the simple similarity, the ray parameter of the intersection point is

$$H_{new} = H_b + (H_a - H_b)\frac{\Delta_a}{\Delta_a - \Delta_b}.$$

Note that we should use Δ_b with negative sign since Δ_b means a signed distance and it is negative.

Checking the height here again, the new point may be either above or below the height field. Replacing the previous point of the same type with the new point, the same procedure can be repeated iteratively.

The secant algorithm is as follows:

```
for (int i = 0; i < SEC_ITER; i++) {
   H = Hb + (Ha-Hb) /(Da-Db) * Da;
   float2 uv = uvin * H + uvout * (1-H);
   D = H - tex2D(hMap, uv).a ;
   if (D < 0) {
      Db = D; Hb = H;
   } else {
      Da = D; Ha = H;
   }
}
```

4.3. Safe iterative methods

Safe methods pay attention to finding the first intersection even if the ray intersects the height field several times. The first algorithm of this section, called linear search is only "quasi-safe". If the steps are larger than the texel size, this algorithm may still fail, but its probability is low. Other techniques, such as the dilation-erosion map, the sphere, cone, and pyramidal mapping are safe, but require preprocessing that generates data encoding the empty spaces in the height field. Note that safe methods guarantee that they do not skip the first intersection, but cannot guarantee that the intersection is precisely located if the number of iterations is limited.

4.3.1. Linear search

Linear search, i.e. *ray-marching* finds a pair of points on the ray that enclose the possibly first intersection, taking steps of the same length on the ray between the entry and exit points (figure 19). Ray marching is an old method of rendering 3D volumetric data [Lev90].

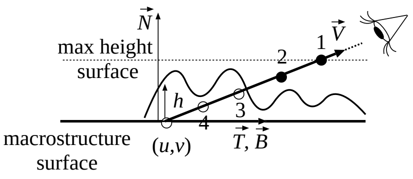


Figure 19: *Linear search that stops at point 3*

If the number of steps between the entry and exit points is `LIN_ITER`, then at a single step the height parameter of the ray, `H`, is decreased by 1/`LIN_ITER`. At each visited point on the ray, the algorithm checks whether or not the point is below the height field. If it is below, then an intersection must exist between the last and the current visited points, thus the linear search can be stopped. Denoting the normalized tangent space entry point by `uvin` and the exit point by `uvout`, the implementation of the linear search phase is as follows:

```
float H = 1.0; // current height
float Hint = 0.0 // height of intersection
for (int i = 0; i < LIN_ITER; i++){
   H -= 1.0f / LIN_ITER;
   uv = uvin * H + uvout * (1-H);
   float h = tex2D(hMap, uv).a;
   if (Hint == 0) // if no point below yet
      if (H <= h) Hint = H; // below
}
// texture coords of intersection
uv = uvin * Hint + uvout * (1-Hint);
```

Note that this algorithm always iterates through the ray between the entry and exit points but stores only the first ray parameter where the ray got below the height field. Continuing the search after finding the first point is not necessary from algorithmic point of view, but GPUs prefer loops of constant number of cycles.

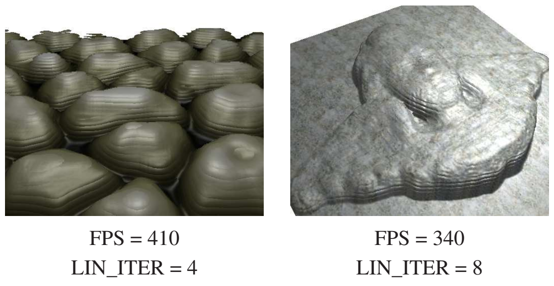


Figure 20: *Linear search. Note the stair-stepping artifacts.*

Linear search was used alone in *Steep parallax mapping* [MM05], and as a first pass of combined methods like *Relief mapping* [POC05] and *Parallax occlusion mapping* [BT04, Tat06b]. Using linear search alone results in stair-stepping artifacts unless the steps are very fine (figure 20). Tatarchuk [Tat06b] solved this problem by adding just a single secant step to a fine linear search. Note that linear search cannot guarantee that no intersection is missed if its steps may skip texels of the height field. The robustness of the algorithm can be increased by making smaller steps, of course, for the price of higher rendering times.

Brawley and Tatarchuk [BT04] noticed that it is worth increasing the number of iteration steps at oblique angles [MM05, Tat06a]. Making the iteration number dependent on the actual pixel requires a dynamic loop supported by Shader Model 3 or higher GPUs. Direct3D 9 instructions computing derivatives, like `tex2D`, force the HLSL compiler to unroll the loop. To avoid unrolling, we can use function `tex2Dlod` setting the mip-map level explicitly to zero or `tex2Dgrad` with explicitly given derivative values [San05].

To make linear search really safe each texel needs to be visited, which is slow if the texture resolution is high. Safe displacement mapping algorithms discussed in the following subsections all target this problem by using some additional data structure encoding empty spaces. Such additional data structures require preprocessing but can always specify the size of step the algorithm may safely take along the ray.

4.3.2. Dilation and erosion maps

Kolb and Rezk-Salama proposed a preprocessing based approach that creates two additional 2D maps with a dilation and erosion process of the original height map [KRS05]. These maps store the minimum-filtered and the maximum-filtered versions of the displacement map specifying a *safety zone*. These zones are used for empty space skipping to find valid ray sections containing all ray surface intersections. Dilation-erosion maps encode the non-empty space by tangent space axis aligned bounding boxes.

4.3.3. Pyramidal displacement mapping

Pyramidal displacement mapping encodes empty spaces in a mip-map like hierarchical scheme. This idea was mentioned in [MM05], relates to dilation and erosion maps as well since the space is subdivided by boxes, and was fully elaborated in [OK06].

A pyramidal displacement map is a quadtree image pyramid. Each leaf texel at the lowest level of the mip-map indicates the difference between the maximum height surface and the current height at the given texel. The root texel of the top mip-map level, on the other hand, denotes the global, minimum difference. This difference is denoted by d_0 in figure 21. Each texel in the inner levels stores the local minimum difference (d_1 in figure 21) of the four children. The image pyramid can be computed on the CPU and on the GPU as well.

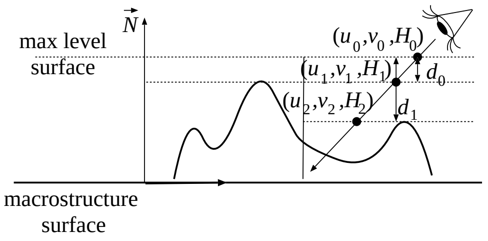


Figure 21: *Pyramidal displacement mapping*

The intersection search starts at the highest level of the pyramidal texture. The top level difference is read. This distance can be safely stepped with height ray parameter H of equation 11 without risking any intersections with the height field. After this step a new sample point is given on the ray. At this point the minimum difference of the next mip-map level is fetched, and the same step is repeated iteratively. When we reach the lowest level, the intersection point is obtained. This algorithm also skips box shaped empty spaces represented by the difference of two subsequent mip-map levels. Note, however, that higher level decisions may get invalid if at a lower level the ray leaves the box below the higher level texel. For example, in figure 21 when the ray crosses the vertical line representing the first level texel boundary, then the decision that the ray height can be decreased by d_1 becomes wrong. Such ray crosses should be detected, and when they occur, the hierarchy traversal should be backtracked and continued with the higher level texel above the crossing ray. Unfortunately, this process makes the otherwise elegant algorithm quite complicated to implement.

4.3.4. Sphere tracing

Sphere tracing was introduced in [Har93] to ray trace implicit surfaces and applied to height field ray tracing in [Don05]. It uses a distance map and an iterative algorithm in order to always find the first hit. A distance map is a 3D

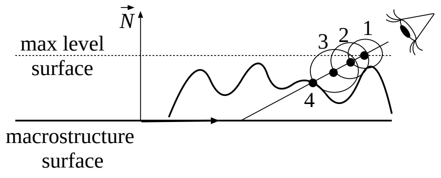


Figure 22: *Displacement mapping with sphere tracing*

texture that is precomputed from 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. The value stored in the texel is the distance from the corresponding point to the closest point on the bumpy surface. This distance can also be interpreted as the radius of the largest sphere centered at the corresponding point, that does not intersect, only touches the height field (figure 22). This property explains the name of the method.

In order to find the first intersection of the ray and the height field, we make safe steps on the ray. When we are at tangent space point `uvh` on the ray, we can safely step with the distance stored in the distance map at this point, not risking that a hill of the height field is jumped over. The distance map is fed to the fragment shader as a monochromatic, 3D texture `distField`:

```
float3 uvh = float3(uvin, 1);
for (int i = 0; i < SPHERE_ITER; i++) {
    float Dist = tex3D(distField, uvh);
    uvh -= Dist * View;
}
```

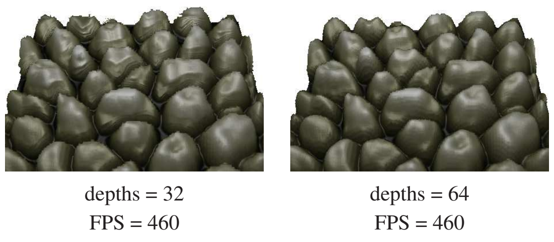


Figure 23: *Sphere tracing results with different distance field texture resolutions. The left image was rendered with a 3D texture of 32 depth layers, while the right image with 64 depth layers.*

Sphere tracing results obtained with different resolution distance maps are shown by figure 23. The 3D distance field texture data may be generated by the *Danielsson's algorithm* [Dan80, Don05]. The distance field generation is quite time consuming. It is worth preparing it only once and storing the result in a file.

4.3.5. Cone stepping

Sphere tracing uses a 3D texture map to represent empty spaces, which poses storage problems, and does not naturally fit to the concept of height fields stored as 2D textures. Based on these observations, *cone tracing* [Dum06] aims to encode empty spaces by a 2D texture having the same structure as the original height map.

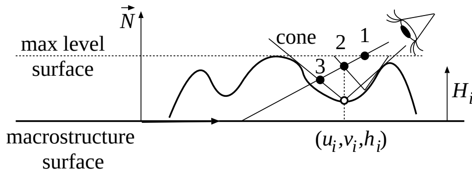


Figure 24: *Displacement mapping with cone stepping*

In cone tracing every height field texel is given a cone that represents the empty space above it (figure 24). The tip of the cone is the heightfield surface at that texel, its axis is parallel to axis z in tangent space, thus the only parameter that needs to be stored to define the cone is its apex angle or the slope of its boundary.

Suppose that the current approximation on the ray is $(u,v,0)+\vec{V}t_i = (u_i,v_i,H_i)$. The height at this point is $h_i = h(u_i,v_i)$. The next approximation,

$$(u_i,v_i,H_i)-\vec{V}\Delta t_i = (u_{i+1},v_{i+1},H_{i+1}),$$

is calculated as the intersection of this ray and the cone standing at (u_i,v_i,h_i). The equation of the cone standing at the height field is

$$(u'-u_i)^2+(v'-v_i)^2 = m^2(h'-h_i)^2,$$

where m equals to the ratio of the cone radius and height, and expresses the apex angle of the cone. Substituting the ray equation into the equation of this cone we obtain

$$(V_x^2+V_y^2)\Delta t_i^2 = m^2(H_i - V_z\Delta t_i - h_i)^2.$$

Solving it for unknown step Δt_i, we get

$$\Delta t_i = \frac{m(H_i-h_i)}{\sqrt{V_x^2+V_y^2}+mV_z}$$

The following shader code uses a 2D map, `ConeMap`, that stores the displacement height values in its red channel and the cone radius per height parameter in its green channel:

```
float3 uvh = float3(uvin, 1);
float lViewxy = length(View.xy);
for (int i = 0; i < CONE_ITER; i++) {
    float h = tex2D(ConeMap, uvh.xy).r;
    float m = tex2D(ConeMap, uvh.xy).g;
    float dts = m/(lViewxy + m * View.z);
    float dt = (uvh.z - h) * dts;
    uvh -= View * dt;
}
```

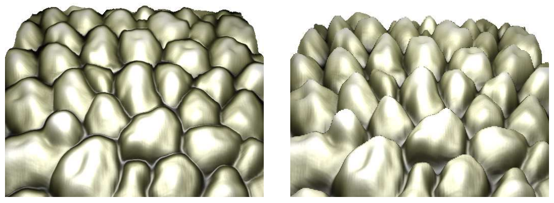


Figure 25: *Cone stepping results with different iteration count setting BIAS* $= 0.7$ *and SCALE* $= 0.3$.

Images rendered with cone stepping are shown by figure 25.

4.4. Combined iterative methods

Combined iterative methods combine a safe or quasi-safe technique to provide robustness and an unsafe method that is responsible for finding the intersection quickly. The safe method should only find a rough approximation of the first intersection and should make sure that from this approximation the unsafe method cannot fail. Table 1 lists iterative displacement mapping methods together with their first reference. The columns correspond to the safe phase, while the rows to the unsafe phase. Methods where a particular phase is missing are shown in the "NO" row or column. Combined methods belong to cells where both phases are defined.

4.4.1. Relief mapping

The pixel shader implementation [POC05, PO05] of *relief mapping* [OBM00, Oli00] uses a two phase root-finding approach to locate the intersection of the height field and the ray. The first phase is a linear search, i.e. ray-marching, which finds a pair of points on the ray that enclose the possibly first intersection. The second phase refines these approximations by a binary search (figure 26).

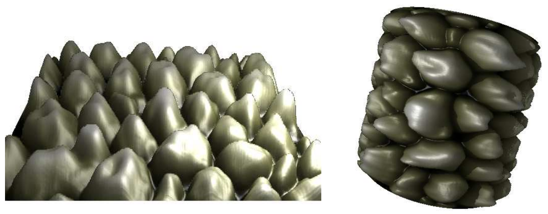


Figure 26: *Relief mapping results (LIN_ITER=* 32 *and BIN_ITER=* 4*)*

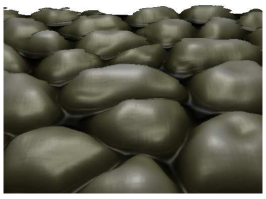
	Ray marching	Empty-space encoding	NO
Binary	Relief mapping [POC05]	Relaxed cone [PO07]	
Secant	DM with ray casting [YJ04] Parallax Occlusion [BT04] Interval mapping [RSP06]		
Slope			Iterative parallax[Pre06]
NO	Steep parallax [MM05]	Sphere tracing [Don05] Erosion-dilation map [KRS05] Pyramidal DM [OK06] Cone stepping [Dum06]	

Table 1: *Classification of iterative displacement mapping (DM) methods. The columns correspond to the first phase (if any) while rows correspond to the second phase.*

4.4.2. Combined linear and secant search

Ray marching can also be improved combined with a series of geometric intersections, according to the *secant method*. The first pass of these algorithms is also a linear search. This kind of combined approach was first proposed by Yerex [YJ04] and independently in *Parallax Occlusion Mapping* [BT04, Tat06a, Tat06b], and showed up in *Interval mapping* [RSP06] too.

The ratio of the number of linear search steps and the number of refinement steps using either binary search or the secant method should be selected carefully. Taking more linear search steps increases robustness, and we should also take into account that a linear search step is executed faster by the graphics hardware. The reason is that the texture coordinates of linear search steps do not depend on the results of previous texture fetches unlike the texture coordinates obtained by binary search or by the secant method, which do depend on the values stored in textures. This dependency is often referred to as *dependent texture fetching*, and results in poorer texture cache utilization and therefore longer latency. It means that the number of secant steps should be small. In practice 1–2 secant steps are sufficient (figure 27).


FPS = 310
LIN_ITER = 4
SEC_ITER = 2

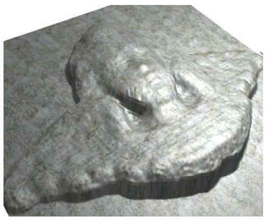

FPS = 225
LIN_ITER = 8
SEC_ITER = 1

Figure 27: *Combined linear and secant search*

We mention that two phase search processes, including a linear search phase and a secant search phase have also proven to be successful in tasks that are independent of displacement mapping, for example in curved reflection computation [UPSK07] and isosurface extraction of volumetric data [HSS*05].

4.4.3. Relaxed Cone Stepping

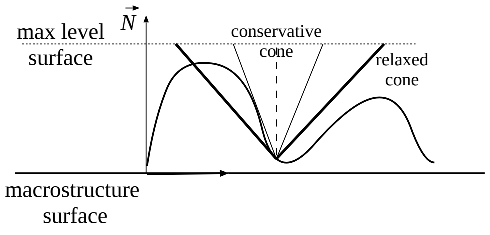


Figure 28: *Difference between conservative and relaxed cones used in cone stepping*

Cone stepping defines a cone with a maximum angle that would not cause the cone to intersect the height field for each texel of the displacement map. This strict rule will cause the rays to stop before reaching the surface, which leads to errors and artifacts if the iteration number is limited. *Relaxed cone stepping* relaxes this rule and defines the cones with the maximum angle that would not cause the view ray to intersect the height field *more than once* [PO07].

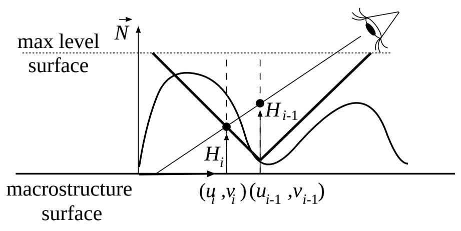


Figure 29: *Relaxed cone stepping*

The intersection search algorithm will be the same as in the case of the original cone stepping algorithm. The final two search results will give an undershooting and an overshooting points. From these points binary [PO07] or secant search can be started to find the exact solution.

5. Silhouette processing

So far we have discussed algorithms that compute the texture coordinates of the visible point assuming a planar macrostructure surface. For triangle meshes, the underlying macrostructure geometry might change and the modified texture coordinates get outside of the texture footprint of the polygon, thus the texture coordinate modification computed with the planar assumption may become invalid. Another problem is that the mesostructure surface may get projected to more pixels than the macrostructure surface, but the fragment shader is invoked only if the macrostructure triangle passes the backface culling, its processed point is projected onto this pixel and passes the early z-test. Thus it can happen that a height map point is ignored because its corresponding point of the macrostructure geometry is not processed by the fragment shader. These issues are particularly important at triangle edges between a front facing and a back facing polygons, which are responsible for the *silhouettes* of the detailed object.

To cope with this problem, a simple silhouette processing approach would discard the fragment if the modified texture coordinate gets outside of texture space of the rendered polygon. While this is easy to test for rectangles, the check becomes more complicated for triangles, and is not robust for meshes that are tessellations of curved surfaces.

For meshes, instead of assuming a planar surface, i.e. a linear approximation, the *local curvature* information should also be considered, and at least a quadratic approximation should be used to describe the surface farther from the processed point. On the other hand, to activate the fragment shader even for those pixels where the mesostructure surface is visible but the macrostructure surface is not, the macrostructure surface should be "thickened" to include also the volume of the height field above it. Of course, the thickened macrostructure surface must also be simple, otherwise the advantages of displacement mapping get lost. Such approaches are discussed in this section.

5.1. Silhouette processing for curved surfaces

In order to get better silhouettes, we have to locally deform the height field according to the curvature of the smooth geometry [OP05]. The mesostructure geometry $\vec{r}(u,v)$ is usually approximated locally by a quadrics [SYL02] in tangent space (compare this second order approximation to the first order approximation of equation 3):

$$\vec{r}(u,v) \approx \vec{q}(u,v) = \vec{r}(u_0,v_0) + [u-u_0, v-v_0] \cdot \begin{bmatrix} \frac{\partial \vec{r}}{\partial u} \\ \\ \frac{\partial \vec{r}}{\partial v} \end{bmatrix} +$$

$$\frac{1}{2} \cdot [u-u_0, v-v_0] \cdot \begin{bmatrix} \frac{\partial^2 \vec{r}}{\partial u^2} & \frac{\partial^2 \vec{r}}{\partial u \partial v} \\ \\ \frac{\partial^2 \vec{r}}{\partial u \partial v} & \frac{\partial^2 \vec{r}}{\partial v^2} \end{bmatrix} \cdot \begin{bmatrix} u-u_0 \\ v-v_0 \end{bmatrix}.$$

where all partial derivatives are evaluated at (u_0,v_0). If (u_0,v_0) corresponds to a vertex of the tessellated mesh, then

$$\vec{r}(u_0,v_0) = \vec{p}(u_0,v_0), \quad \frac{\partial \vec{r}}{\partial u} = \vec{T}, \quad \frac{\partial \vec{r}}{\partial v} = \vec{B}.$$

Matrix

$$\mathbf{H} = \begin{bmatrix} \frac{\partial^2 \vec{r}}{\partial u^2} & \frac{\partial^2 \vec{r}}{\partial u \partial v} \\ \\ \frac{\partial^2 \vec{r}}{\partial u \partial v} & \frac{\partial^2 \vec{r}}{\partial v^2} \end{bmatrix}$$

is a *Hessian matrix*. The eigenvalues of this matrix correspond to the *principal curvatures* and the eigenvectors to the *principal curvature directions*.

The second order approximation is the sum of the linear form represented by the macrostructure mesh and a quadratic form

$$\vec{q}(u,v) \approx \vec{p}(u,v) + \frac{1}{2} \cdot [u-u_0, v-v_0] \cdot \mathbf{H} \cdot [u-u_0, v-v_0]^T.$$

Let us transform this quadratic surface to tangent space, and consider only the third coordinate h_q representing the height function of the quadratic surface:

$$h_q(u,v) = (\vec{q}(u,v) - \vec{p}(u,v)) \cdot \vec{N}^0 =$$

$$a(u-u_0)^2 + b(u-u_0)(v-v_0) + c(v-v_0)^2. \qquad (13)$$

The elements of the Hessian matrix and parameters a,b,c can be computed analytically if the parametric equation of that surface which has been tessellated is known (this is the case if we tessellate a sphere, a cylinder, etc.). If only the tessellated surface is available, then the derivatives should be determined directly from the mesh around vertex $\vec{p}_0$. Let us suppose that the vertices of the triangles incident to $\vec{p}_0$ are $(u_0,v_0,h_0), (u_1,v_1,h_1), \ldots, (u_n,v_n,h_n)$ in the tangent space attached to $\vec{p}_0$. Substituting these points to the quadratic approximation we obtain for $i = 1,\ldots,n$

$$h_i \approx a(u_i-u_0)^2 + b(u_i-u_0)(v_i-v_0) + c(v_i-v_0)^2.$$

since $h_0 = 0$ in the tangent space of $\vec{p}_0$. This is a linear system

$$\mathbf{e} = \mathbf{A} \cdot \mathbf{x},$$

where $\mathbf{e} = [h_1,\ldots,h_n]^T$ is an n-element vector,

$$\mathbf{A} = \begin{bmatrix} (u_1-u_0)^2, & (u_1-u_0)(v_1-v_0), & (v_1-v_0)^2 \\ \vdots & \vdots & \vdots \\ (u_n-u_0)^2, & (u_n-u_0)(v_n-v_0), & (v_n-v_0)^2 \end{bmatrix}$$

is an n-row, 3-column matrix, and

$$\mathbf{x} = \begin{bmatrix} a \\ b \\ c \end{bmatrix}.$$

is a three-element vector. Note that the number of equations is $n > 3$ and just three unknowns exist, thus the equation is overdetermined. An approximate solution with minimal least-squares error can be obtained with the pseudo inverse method:

$$\mathbf{x} = (\mathbf{A}^T \cdot \mathbf{A})^{-1} \cdot \mathbf{A}^T \cdot \mathbf{e}.$$

Note that just a 3×3 matrix needs to be inverted.

Having determined parameters a, b, c the height of the quadratic approximation of the smooth surface is obtained using equation 13.

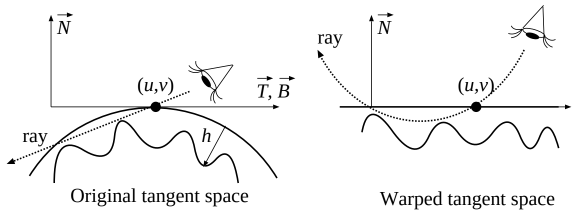


Figure 30: *Silhouette processing with tangent space warping*

The height function of the bumpy surface is approximately $h_q(u,v) + h(u,v)$. Note that this is only an approximation since we should have moved the height into the direction of the normal vector of the quadratic surface, which is not constant any more. However, for smaller curvatures, this approximation is acceptable. To make sure that the rasterizer does not miss fragments, $h(u,v)$ must be negative.

The ray tracing scenario is shown by figure 30. In order to trace back this problem to the previous ones where the macrostructure surface is flat, let us apply a non-linear *warping* to the whole space, which flattens the quadratic surface to a plane. The required warping at (u,v) is $-h_q(u,v)$. This warping transforms the ray as well, so the original ray equation $ray(t) = (u,v,0) - \vec{V}t$ is modified to

$$ray'(t) = (u,v,0) - t\vec{V} - h_q(u - V_x t, v - V_y t) =$$

$$(u,v,0) - t\vec{V} - t^2\vec{N}(aV_x^2 + bV_xV_y + cV_y^2).$$

It means that we can use ray marching, but the steps should be updated according to this formula. When the z coordinate of the ray gets negative, it leaves the space of possible intersections. If we have not found any intersection so far, then this fragment should be discarded.

5.2. Thick surfaces

To activate the fragment shader even for those pixels where the mesostructure surface is visible but the macrostructure surface is not, the macrostructure surface is thickened and each triangle is handled like a three-dimensional volume that includes the height field. The geometric representation of these volumes can be kept simple if we represent them by the boundary surfaces of their bounding volumes. A particularly popular and effective choice is the *prism* shaped bounding volume of the mesostructure surface above each macrostructure triangle. The prism is obtained by extruding the base triangle along the vertex normals with the maximum height. Each prism is rendered by sending its eight face triangles through the pipeline. Since these prisms include the height field, rasterizing the face triangles guarantees that the fragment shader is activated for all pixels in which the height field is visible.

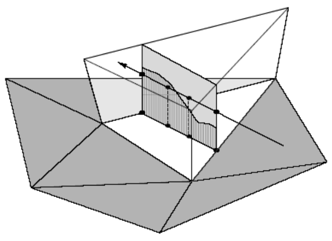


Figure 31: *Sampling within the extruded prism with a slice of the displacement map*

In Hirche's *local ray tracing* method [HEGD04] during the rasterization of the prism faces ray marching is used to detect an intersection with the displaced surface. At each sample point on the ray, the height of the sampling position is compared to the height field (figure 31). If an intersection is detected, the resulting fragment is shaded, its depth is re-evaluated, and the fragment is written into the framebuffer with its correct z-value. The original algorithm executed the prism construction on the CPU, but in Shader Model 4 GPUs the geometry shader can also do that. The ray segment where the marching algorithm should run is bounded by the points where the ray enters and exits the prism. The entry position is given by rasterization, but the computation of the exit point is not so straightforward. Hirche used an adaptation of the *Projected tetrahedra algorithm* by Shirley and Tuchman [ST90], while Wang et al. [WTL*04] computed ray-plane intersections for the current view ray to find the exit point.

Another problem that can occur while extruding prisms is that the vertices defining one side of the prism are not necessary coplanar. This can cause problems if two adjacent prisms split their neighboring sides differently into two triangles, which can lead to cracks. This problem can be solved with the correct ordering of the new vertices of the prism faces.

Dufort et al. used a voxel traversal algorithm inside the bounding prism [DLP05]. This method can simulate displacement maps as well as semitransparent surface details, including several surface shading effects (visual masking, self-shadowing, absorption of light) at interactive rates.

The idea of mapping a three-dimensional volume onto a surface with prisms also showed up in *shell maps* [PBFJ05].

5.2.1. Shader Model 4 adaptation of the local ray tracing method

The displacement techniques using bounding prisms can be implemented purely in hardware with the geometry shader announced in Shader Model 4. One possible Diret3D 10 implementation can be found in the DirectX SDK 2006 August.

The prism extrusion and the ray exit point calculation are done in the geometry shader. The geometry shader reads the triangle data (three vertices) and creates additional triangles forming a prism with edge extrusion. It splits the prism into three tetrahedra just like in [HEGD04]. For each vertex of a tetrahedron, the distance from the “rear” of the tetrahedron is computed. Then the triangles with these additional data are passed to the clipping and rasterizer units. From the rasterizer the fragment shader gets the entry point and its texture coordinate, as well as the exit point and its texture coordinates, which are interpolated by the rasterizer from the distance values computed by the geometry shader.

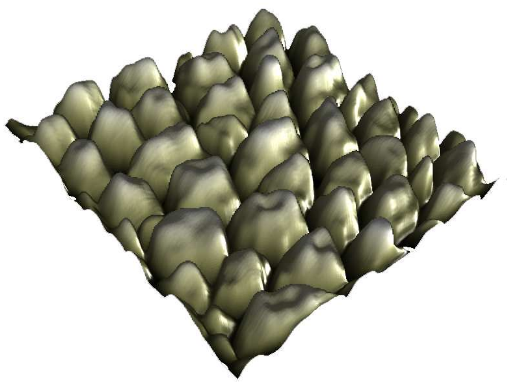


Figure 32: *Displacement mapping with local ray tracing implemented on the geometry shader of a NVidia 8800 GPU and rendered at 200 FPS*

Figure 32 has been rendered with this algorithm on a Shader Model 4 compatible GPU.

6. Self-shadowing computation

Self-shadowing can be easily added to iterative fragment shader methods. By calling it with the Light vector (pointing from the shaded point to the light source) instead of the View vector, it is possible to decide whether the shaded point is in shadow or not (figure 33). Using more than one light vector per light source, even soft shadows can be generated [Tat06a, Tat06b].

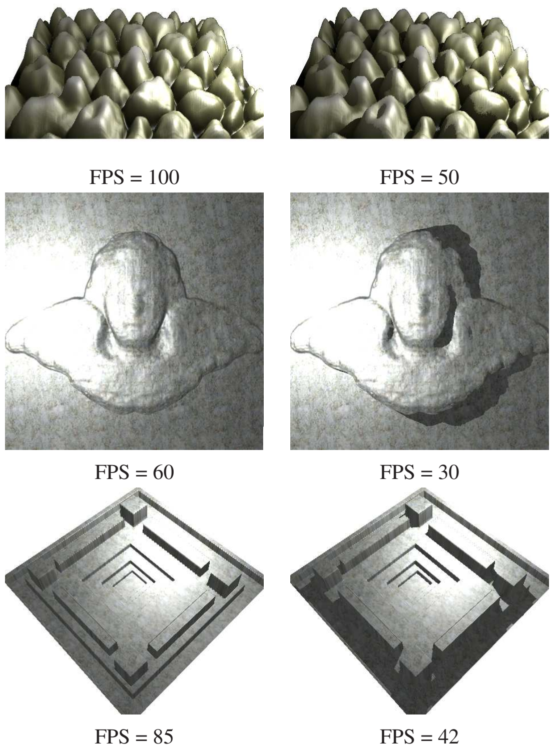


Figure 33: *The secant method without (left) and with self-shadowing (right) setting LIN_ITER* $= 32$*, SEC_ITER* $= 4$

We note that shadows can also be burnt into the information associated with the texels, as happens in the case of *bi-directional texture functions*. Bi-directional texture functions have been introduced by Dana et al. [DGNK97, SBLD03, MGW01, HDKS00]. An excellent review of the acquisition, synthesis, and rendering of bi-directional texture functions can be found in [MMS*04].

6.1. Horizon mapping

Another fast method for self-shadowing computation is *horizon mapping* [Max88]. The idea behind horizon mapping is to precompute the angle to the horizon in a discrete number of directions. Horizon angles represent at what height the sky becomes visible at the given direction (i.e. passes over the horizon). This parametrization can be used to produce the self-shadowing of geometry. An extension of this algorithm also considered the local geometry [SC00].

6.2. Displacement mapping with precomputed visibility

Heidrich et al. introduced a method similar to horizon mapping [HDKS00], but instead of storing zenith angles, they stored the visible points at predefined directions. If we assume that the height map is attached to a specific, fixed base geometry, we can precompute for each point on the height field and for each discrete direction the point which is hit by this ray. Since this intersection point is some point in the same height field, it is unambiguously characterized by a 2D texture coordinate. By chaining together this visibility information, we can also generate a multitude of different light paths for computing the indirect illumination in the height field.

To determine if a given point lies in the shadow for some light direction, we can simply find the closest direction and check whether a height field point is stored for this ray. For a higher quality test, we can easily determine the three directions that are close to the light direction, and then interpolate the visibility values. This yields a visibility factor between 0 and 1 defining a smooth transition between light and shadow.

6.3. View dependent displacement mapping

View dependent displacement mapping [WWT*04] also takes a pre-computation approach. While this method provides the highest quality results, handles shadows and curved surfaces as well, it also requires considerable preprocessing time and storage space.

This method takes pre-defined view rays and computes the difference of the base surface and the visible point for each texel and for each view ray. To handle curved surfaces, this operation is repeated for a couple of curvature values defined along the view ray.

The results of this precomputation is a 5D function called *View dependent displacement map* (*VDM*), which can be interpreted as an array of distances $d_{VDM}[u,v,\theta,\phi,c]$. Indices of this array are the u,v texture coordinates, the θ,ϕ spherical coordinates of the view direction in tangent frame, and curvature c along the view direction.

When a texel (u,v) is processed, first the curvature of the mean surface along the view direction is determined:

$$c = \frac{c_{max}(\vec{D}_{max}\cdot\vec{V})^2 + c_{min}(\vec{D}_{min}\cdot\vec{V})^2}{1-(\vec{N}\cdot\vec{V})^2}$$

where c_{max} and c_{min} are the *principal curvature values* corresponding to *principal curvature directions* $\vec{D}_{max}$ and $\vec{D}_{min}$, respectively. The θ,ϕ values can be obtained from viewing vector $\vec{V}$. Having these, the new texture coordinates are

$$(u',v') = (u,v) + d_{VDM}[u,v,\theta,\phi,c](V_x,V_y).$$

In order to cope with the high storage requirements, *singular value decomposition* can be used to compress the 5D function and replace it by lower dimensional tables.

In [WD05] a compression method is presented that combines k-means clustering and non-negative matrix factorization. Wang et al. introduced an extension of View dependent displacement maps, called *Generalized displacement maps* (*GDM*) [WTL*04]. This method can model surfaces with more than one offset value per texel, thus arbitrary non-height-field mesostructure. Furthermore, the GDM approach overcomes both the texture distortion problems of VDM and the computational expense of directly mapped geometry by computing visibility jointly in texture space and object space.

7. Conclusions

In this paper we reviewed vertex shader and fragment shader solutions of displacement mapping. Their comparative advantages and disadvantages depend on the properties of the scene, including the number of triangles that are subject to displacement mapping and the number of pixels covered by them.

The complexity, i.e. the rendering time, of vertex shader based approaches depends on the number of triangles generated by displacement mapping. This number corresponds to the resolution of the height map. A 256×256 resolution height map generates 128 thousand triangles when mapped onto an object. Thus if there are many objects in the scene (including those that are not even visible and are eliminated by the graphics pipeline) and we use high resolution height maps, then vertex shader methods get slower. To compensate this, view and visibility culling [WB05] is worth using and we should not send the surely invisible objects to the rendering pipeline. On the other hand, level of detail techniques may be introduced. We may maintain several versions of the same height field (mip-mapping) and of the same object tessellated on different levels, and render one object and mip-map level corresponding to the distance from the camera. Note that this decision and the level of detail selection must be done by the CPU, because the GPU cannot alter the tessellation level (this changes from Shader Model 4). It is widely believed that vertex shader approaches are slower than fragment shader approaches since GPUs have more pixel-processing power than vertex-processing power, and pixel shaders are better equipped to access textures. According to our tests, however, this is not the case anymore. We think that this belief was born when the vertex shaders were not able to read textures, but those times are gone.

Per vertex displacement mapping really changes the geometry, thus it can handle cases curved surfaces and can provide correct *silhouettes* automatically. They can be seamlessly incorporated into depth mapped shadow algorithms, but shadow volumes require geometry shader or complicated vertex shader approaches. Vertex shader solutions are relatively easy to implement. Summarizing, vertex shader solutions are the winners if only a part of the scene is displace-

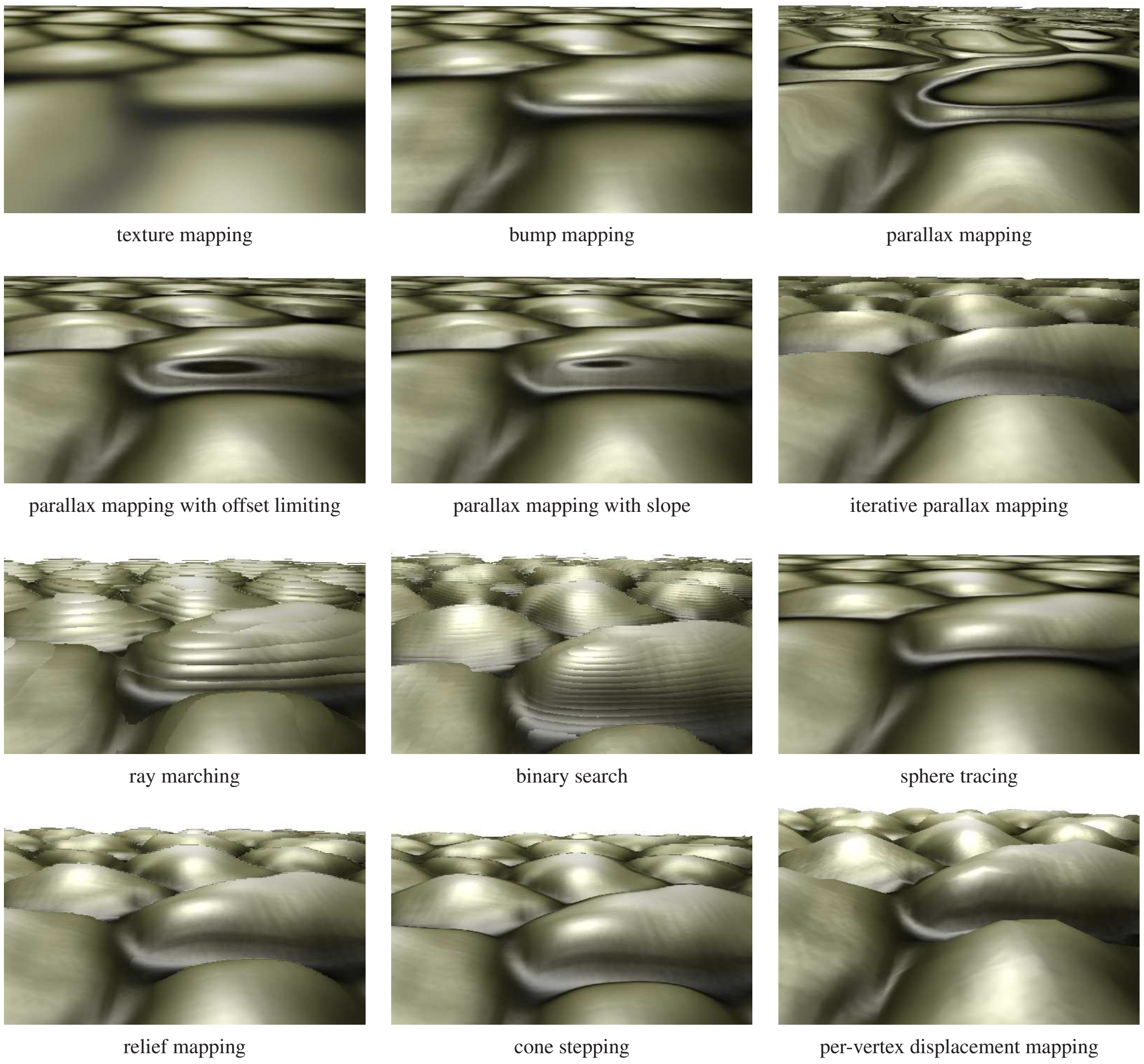

texture mapping
bump mapping
parallax mapping
parallax mapping with offset limiting
parallax mapping with slope
iterative parallax mapping
ray marching
binary search
sphere tracing
relief mapping
cone stepping
per-vertex displacement mapping

Figure 34: *Comparison of mapping techniques assuming normal displacement sizes*

ment mapped, or we apply view culling or level of detail techniques, or if the displacements are really large.

The rendering time of fragment shader approaches depends on the number of pixels covering the displacement mapped objects. Thus if we usually get far from these objects or the objects are not visible, then fragment shader methods are faster than vertex shader techniques.

Fragment shader methods are approximate by nature, although different algorithms take different compromises between accuracy and speed. An accurate solution that handles silhouettes as well is quite difficult to implement, and the implementation is rather slow. Thus fragment shader solutions are viable if the height map has high frequency characteristics, and thus has high resolution, and the objects are small (i.e. usually cover just a smaller part of the screen), and the displacements are also small, so approximation errors are not so visible.

Displacement mapping also has limitations. For example, it is constrained to a one-valued height field so holes cannot be added. This restriction is addressed by *Deformation displacement mapping* [Elb02, SKE05] and *Generalized displacement mapping* [WTL*04]. However, despite to its limitations, it has become a popular technique and we can expect its increasing role in games and real-time systems.

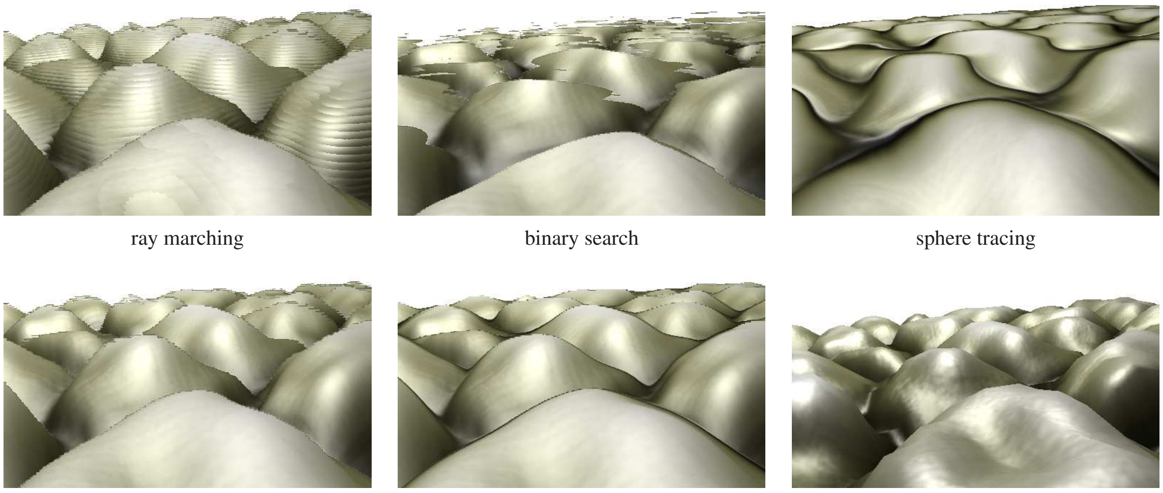


Figure 35: *Comparison of high quality mapping techniques assuming extreme displacement sizes*

8. Acknowledgement

This work has been supported by the GameTools FP6 (IST-2-004363) project, Hewlett-Packard, and the National Office for Research and Technology (Hungary).

References

[APS00] Ashikhmin M., Premoze S., Shirley P.: A microfacet-based BRDF generator. In *SIGGRAPH 2000 Proceedings* (2000), Akeley K., (Ed.), pp. 65–74.

[ATI03] ATI: Normalmapper tool, 2003. Available: http://www2.ati.com/developer/NormalMapper-3_2_2.zip.

[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.

[Bla92] Blasco O.: Curvature simulation using normal maps. In *Game Programming Gems III*, Treglia D., (Ed.). Charles River Media, Boston, 1992, pp. 433–443.

[Bli77] Blinn J. F.: Models of light reflection for computer synthesized pictures. In *Computer Graphics (SIGGRAPH '77 Proceedings)* (1977), pp. 192–198.

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

[Bly06] Blythe D.: The Direct3D 10 system. In *SIGGRAPH 2006 Proceedings* (2006), pp. 724–734.

[BM93] Becker B. G., Max N. L.: Smooth transitions between bump rendering algorithms. In *SIGGRAPH '93 Proceedings* (1993), ACM Press, pp. 183–190.

[BN76] Blinn J. F., Newell M. E.: Texture and reflection in computer generated images. *Communications of the ACM 19*, 10 (1976), 542–547.

[BS05] Boubekeur T., Schlick C.: Generic mesh refinement on GPU. In *HWWS '05: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (2005), ACM Press, pp. 99–104.

[BT04] Brawley Z., Tatarchuk N.: Parallax occlusion mapping: Self-shadowing, perspective-correct bump mapping using reverse height map tracing. In *ShaderX3: Advanced Rendering Techniques in DirectX and OpenGL*, Engel W., (Ed.). Charles River Media, Cambridge, MA, 2004.

[Cat74] Catmull E.: *A Subdivision Algorithm for Computer Display and Curved Surfaces*. PhD thesis, University of Utah, Salt Lake City, Utah, 1974. Ph.D. Dissertation.

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

[CG85] Carey R. J., Greenberg D. P.: Textures for realistic image synthesis. *Computers and Graphics 9*, 2 (1985), 125–138.

[CMSR98] Cignoni P., Montani C., Scopigno R., Rocchini C.: A general method for preserving attribute values on simplified meshes. In *IEEE Visualization* (1998), pp. 59–66.

[Coo84] Cook R. L.: Shade trees. In *SIGGRAPH '84 Proceedings* (1984), ACM Press, pp. 223–231.

[CORLS96] Cohen-Or D., Rich E., Lerner U., SHENKAR V.: A Real-Time Photo-Realistic Visual Fly-through. *IEEE Transactions on Visualization and Computer Graphics 2*, 3 (1996), 255–264.

[Cro77] CROW F. C.: Shadow algorithm for computer graphics. In *Computer Graphics (SIGGRAPH '77 Proceedings)* (1977), pp. 242–248.

[CT81] COOK R., TORRANCE K.: A reflectance model for computer graphics. *Computer Graphics 15*, 3 (1981), 7–24.

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

[DGNK97] DANA K., GINNEKEN B., NAYAR S., KOENDERINK J.: Reflectance and texture of real-world surfaces. *ACM Transactions on Graphics 18*, 1 (1997), 1–34.

[DH00] DOGGETT M., HIRCHE J.: Adaptive view dependent tessellation of displacement maps. In *HWWS '00: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware* (2000), ACM Press, pp. 59–66.

[Die00] DIETRICH S.: *Elevation Maps*. NVIDIA Corporation, 2000.

[DKS01] DOGGETT M. C., KUGLER A., STRASSER W.: Displacement mapping using scan conversion hardware architectures. *Computer Graphics Forum 20*, 1 (2001), 13–26.

[DLP05] DUFORT J.-F., LEBLANC L., POULIN P.: Interactive rendering of meso-structure surface details using semi-transparent 3d textures. In *Proc. Vision, Modeling, and Visualization 2005* (2005), pp. 399–406.

[Don05] DONELLY W.: Per-pixel displacement mapping with distance functions. In *GPU Gems 2*, Parr M., (Ed.). Addison-Wesley, 2005, pp. 123–136.

[Dum06] DUMMER J.: *Cone Step Mapping: An Iterative Ray-Heightfield Intersection Algorithm*. Tech. rep., 2006. http://www.lonesock.net/files/ConeStepMapping.pdf.

[EBAB05] ESPINO F. J., BÓO M., AMOR M., BRUGUERA J. D.: Adaptive tessellation of bézier surfaces based on displacement maps. In *The 13-th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision* (2005), pp. 29–32.

[Elb02] ELBER G.: Geometric deformation-displacement maps. In *PG '02: Proceedings of the 10th Pacific Conference on Computer Graphics and Applications* (Washington, DC, USA, 2002), IEEE Computer Society, p. 156.

[EY03] ELHELW M. A., YANG G.-Z.: Cylindrical relief texture mapping. *Journal of WSCG 11* (2003), 125–132.

[Fou92] FOURNIER A.: *Filtering Normal Maps and Creating Multiple Surfaces*. Tech. rep., Vancouver, BC, Canada, 1992.

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

[GH99] GUMHOLD S., HÜTTNER T.: Multiresolution rendering with displacement mapping. In *HWWS '99: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware* (1999), ACM Press, pp. 55–66.

[Har93] HART J. C.: Sphere tracing: Simple robust antialiased rendering of distance-based implicit surfaces. In *SIGGRAPH 93 Course Notes: Modeling, Visualizing, and Animating Implicit Surfaces*. 1993, pp. 14–1 to 14–11.

[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.

[Hec86] HECKBERT P. S.: Survey of texture mapping. *IEEE Computer Graphics and Applications 6*, 11 (1986), 56–67.

[HEGD04] HIRCHE J., EHLERT A., GUTHE S., DOGGETT M.: Hardware accelerated per-pixel displacement mapping. In *Proceedings of Graphics Interface* (2004), pp. 153–158.

[HLHS03] HASENFRATZ J.-M., LAPIERRE M., HOLZSCHUCH N., SILLION F. X.: A survey of realtime soft shadow algorithms. In *Eurographics Conference. State of the Art Reports* (2003).

[HS98] HEIDRICH W., SEIDEL H.-P.: Ray-tracing procedural displacement shaders. In *Proceedings of Graphics Interface* (1998), pp. 8–16.

[HS04] HENNING C., STEPHENSON P.: Accelerating the ray tracing of height fields. In *GRAPHITE '04: Proceedings of the 2nd international conference on Computer graphics and interactive techniques in Australia and South East Asia* (2004), ACM Press, pp. 254–258.

[HSS*05] HADWIGER M., SIGG C., SCHARSACH H., BÜHLER K., GROSS M.: Real-time ray-casting and advanced shading of discrete isosurfaces. In *Eurographics Conference* (2005), pp. 303–312.

[HTSG91] HE X., TORRANCE K., SILLION F., GREENBERG D.: A comprehensive physical model for light reflection. *Computer Graphics 25*, 4 (1991), 175–186.

[JWP05] JESCHKE S., WIMMER M., PURGATHOFER W.: Image-based representations for accelerated rendering of complex scenes. In *Eurographics, 2005. State of the Art Reports* (2005), p. 61.

[Kil00] KILGARD M.: A practical and robust bump-mapping technique for today's GPU's, 2000. http://www.nvidia.com/.

[KKI*01] KANEKO T., KAKAHEI T., INAMI M., KAWAKAMI N., YANAGIDA Y., MAEDA T., TACHI S.:

Detailed shape representation with parallax mapping. In *Proceedings of ICAT 2001* (2001), pp. 205–208.

[KRS05] KOLB A., REZK-SALAMA C.: Efficient empty space skipping for per-pixel displacement mapping. In *Vision, Modeling and Visualization* (2005).

[KS01] KAUTZ J., SEIDEL H.-P.: Hardware accelerated displacement mapping for image based rendering. In *Proceedings of Graphics Interface* (2001), pp. 61–70.

[KSK01] KELEMEN C., SZIRMAY-KALOS L.: A microfacet based coupled specular-matte BRDF model with importance sampling. In *Eurographics 2001, Short papers, Manchester* (2001), pp. 25–34.

[Lev90] LEVOY M.: Efficient ray tracing of volume data. *ACM Transactions on Graphics 9*, 3 (1990), 245–261.

[LKG*03] LENSCH H. P. A., KAUTZ J., GOESELE M., HEIDRICH W., SEIDEL H.-P.: Image-based reconstruction of spatial appearance and geometric detail. *ACM Transactions on Graphics 22*, 2 (2003), 234–257.

[LP95] LOGIE J. R., PATTERSON J. W.: Inverse displacement mapping in the general case. *Computer Graphics Forum 14*, 5 (December 1995), 261–273.

[LPFH01] LENGYEL J., PRAUN E., FINKELSTEIN A., HOPPE H.: Real-time fur over arbitrary surfaces. In *Proceedings of the 2001 Symposium on Interactive 3D Graphics* (2001), ACM Press, pp. 227–232.

[LS95] LEE C.-H., SHIN Y. G.: An efficient ray tracing method for terrain rendering. In *Pacific Graphics '95* (1995), pp. 183–193.

[Max88] MAX N.: Horizon mapping: Shadows for bump-mapped surfaces. *The Visual Computer 4*, 2 (1988), 109–117.

[MGW01] MALZBENDER T., GELB D., WOLTERS H.: Polynomial texture maps. In *SIGGRAPH 2001 Proceedings* (2001), pp. 519–528.

[MJW07] MANTLER S., JESCHKE S., WIMMER M.: *Displacement Mapped Billboard Clouds*. Tech. Rep. TR-186-2-07-01, Jan. 2007.

[MM02] MOULE K., MCCOOL M.: Efficient bounded adaptive tessellation of displacement maps. In *Proceedings of the Graphics Interface* (2002), pp. 171–180.

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

[MMS*04] MUELLER G., MESETH J., SATTLER M., SARLETTE R., KLEIN R.: Acquisition, synthesis and rendering of bidirectional texture functions. In *Eurographics 2004, State of the art reports* (2004).

[Mus88] MUSGRAVE F. K.: Grid tracing: Fast ray tracing for height fields. *Research Report YALEU/DCS/RR-639* (1988).

[OB99] OLIVEIRA M. M., BISHOP G.: *Relief Textures*. Tech. rep., Chapel Hill, NC, USA, 1999.

[OBM00] OLIVEIRA M. M., BISHOP G., MCALLISTER D.: Relief texture mapping. In *SIGGRAPH 2000 Proceedings* (2000), pp. 359–368.

[OK06] OH K., KI H.: Pyramidal displacement mapping: A GPU-based atrifacts-free ray tracing through an image pyramid. In *ACM Symposium on Virtual Reality Software and Technology (VRST'06)* (2006), pp. 75–82.

[Oli00] OLIVEIRA M. M.: *Relief Texture Mapping*. PhD thesis, University of North Carolina, 2000.

[OP05] OLIVEIRA M. M., POLICARPO F.: *An Efficient Representation for Surface Details*. Tech. rep., 2005. UFRGS Technical Report RP-351.

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

[PBFJ05] PORUMBESCU S. D., BUDGE B., FENG L., JOY K. I.: Shell maps. In *SIGGRAPH 2005 Proceedings* (2005), pp. 626–633.

[PH96] PHARR M., HANRAHAN P.: Geometry caching for ray-tracing displacement maps. In *Eurographics Rendering Workshop 1996 Proceedings* (1996), Springer Wien, pp. 31–40.

[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.

[PO07] POLICARPO F., OLIVEIRA M. M.: Relaxed cone stepping for relief mapping. In *GPU Gems 3*, Nguyen H., (Ed.). Addison Wesley, 2007.

[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.

[Pre06] PREMECZ M.: Iterative parallax mapping with slope information. In *Central European Seminar on Computer Graphics* (2006). http://www.cescg.org/CESCG-2006/papers/TUBudapest-Premecz-Matyas.pdf.

[PRZ05] PARILOV E., ROSENBERG I., ZORIN D.: *Real-time rendering of normal maps with discontinuities*. Tech. rep., 2005. CIMS.

[PS02] PARILOV S., STÜRZLINGER W.: Layered relief textures. In *WSCG Proceedings* (2002), pp. 357–364.

[QQZ*03] QU H., QIU F., ZHANG N., KAUFMAN A., WAN M.: Ray tracing height fields. In *Computer Graphics International* (2003), pp. 202–209.

[RSP06] RISSER E. A., SHAH M. A., PATTANAIK S.: Interval mapping poster. In *Symposium on Interactive 3D Graphics and Games (I3D)* (2006).

[RTG97] RUSHMEIER H., TAUBIN G., GUEZIEZ A.: Applying shape from lighting variation to bump map capture. In *Proc. 8th Eurographics Rendering Workshop* (1997), pp. 35–44.

[San05] SANDER P.: DirectX9 High Level Shading Language. In *Siggraph 2005 Tutorial* (2005). http://ati.amd.com/developer/SIGGRAPH05/ Shading-Course_HLSL.pdf.

[SBLD03] SUYKENS F., BERGE K., LAGAE A., DUTRÉ P.: Interactive rendering with bidirectional texture functions. *Computer Graphics Forum (Eurographics 03) 22*, 3 (2003), 464–472.

[SC00] SLOAN P., COHEN M.: Interactive horizon mapping. In *Rendering Techniques '00 (Proc. Eurographics Workshop on Rendering)* (2000), pp. 281–286.

[Sch94] SCHLAG J.: Fast embossing effects on raster image data. In *Graphics Gems IV*, Heckbert P., (Ed.). Academic Press, 1994, pp. 433–437.

[Sco07] SCOTT W.: *Nvidia's GeForce 8800 graphics processor*. Tech. rep., 2007. http://techreport.com/articles.x/11211/1.

[Sen04] SEN P.: Silhouette maps for improved texture magnification. In *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (New York, NY, USA, 2004), ACM Press, pp. 65–73.

[SKALP05] SZIRMAY-KALOS L., ASZÓDI B., LAZÁNYI I., PREMECZ M.: Approximate ray-tracing on the GPU with distance impostors. *Computer Graphics Forum 24*, 3 (2005), 695–704.

[SKE05] SCHEIN S., KARPEN E., ELBER G.: Real-time geometric deformation displacement maps using programmable hardware. *The Visual Computer 21*, 8–10 (2005), 791–800.

[SSS00] SMITS B. E., SHIRLEY P., STARK M. M.: Direct ray tracing of displacement mapped triangles. In *Proceedings of the Eurographics Workshop on Rendering Techniques 2000* (2000), Springer-Verlag, pp. 307–318.

[ST90] SHIRLEY P., TUCHMANN A.: A polygonal approximation to direct scalar volume rendering. *ACM Computer Graphics 24* (1990), 63–70.

[SYL02] SYLVAIN P.: A survey of methods for recovering quadrics in triangle meshes. *ACM Computing Surveys 34*, 2 (2002), 1–61.

[Tai92] TAILLEFER F.: Fast inverse displacement mapping and shading in shadow. In *Graphics Interface '92 Workshop on Local Illumination* (1992), pp. 53–60.

[Tat06a] TATARCHUK N.: Dynamic parallax occlusion mapping with approximate soft shadows. In *SI3D '06: Proceedings of the 2006 symposium on Interactive 3D graphics and games* (2006), ACM Press, pp. 63–69.

[Tat06b] TATARCHUK N.: Practical parallax occlusion mapping with approximate soft shadows for detailed surface rendering. In *ShaderX 5*, Engel W., (Ed.). Charles River Media, Boston, 2006, pp. 75–105.

[TC05] TARINI M., CIGNONI P.: Pinchmaps: Textures with customizable discontinuities. *Computer Graphics Forum 24*, 3 (2005), 557–568.

[TCRS00] TARINI M., CIGNONI P., ROCCHINI C., SCOPIGNO R.: Real time, accurate, multi-featured rendering of bump mapped surfaces. In *Eurographics Conference* (2000), vol. 19, 3 of *Computer Graphics Forum*, pp. 119–130.

[UPSK07] UMENHOFFER T., PATOW G., SZIRMAY-KALOS L.: Robust multiple specular reflections and refractions. In *GPU Gems 3*, Nguyen H., (Ed.). Addison Wesley, 2007, pp. 317–338.

[WB05] WIMMER M., BITTNER J.: Hardware occlusion queries made useful. In *GPU Gems 2*. Addison-Wesley, 2005, pp. 91–108.

[WD05] WANG J., DANA K. J.: Compression of view dependent displacement maps. In *Texture 2005: Proceedings of the 4th International Workshop on Texture Analysis and Synthesis* (2005), pp. 143–148.

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

[WMF*00] WANG X. C., MAILLOT J., FIUME E., NG-THOW-HING V., WOO A., BAKSHI S.: Feature-based displacement mapping. In *Proceedings of the 2000 Eurographics Workshop on Rendering Techniques* (2000), pp. 257–268.

[WSP04] WIMMER M., SCHERZER D., PURGATHOFER W.: Light space perspective shadow maps. In *Eurographics Symposium on Rendering* (2004), pp. 143–151.

[WTL*04] WANG X., TONG X., LIN S., HU S., GUO B., SHUM H.-Y.: Generalized displacement maps. In *Proceedings of the 2004 Eurographics Symposium on Rendering* (2004), pp. 227–234.

[WWT*04] WANG L., WANG X., TONG X., LIN S., HU S., GUO B.: View-dependent displacement mapping. *ACM Transactions on Graphics 22*, 3 (2004), 334–339.

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

[ZTCS99] ZHANG R., TSAI P.-S., CRYER J. E., SHAH M.: Shape from shading: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence 21*, 8 (1999), 690–706.