2.5

Parallax Occlusion Mapping: Self-Shadowing, Perspective-Correct Bump Mapping Using Reverse Height Map Tracing

Zoe Brawley and Natalya Tatarchuk

Introduction

Throughout history artists specialized in creating the illusion of detail and depth without actually building a concrete model of reality on the canvas. Similarly, in computer graphics we frequently want to create a compelling impression of the scene without the full cost of highly detailed geometry. To avoid creating extremely high-polygon count models, bump mapping was introduced in early computer graphics days in [Blinn78]. Bump mapping is a technique for making surfaces appear detailed and uneven in some pre-determined manner by perturbing the surface normal using a texture. This approach creates the visual illusion of surface detail that would otherwise eat up most of the project's polygonal budgets such as fissures and cracks in terrain and rocks, textured bark on trees, clothes, wrinkles, etc. Since then there have been many extensions to the basic bump mapping technique including emboss bump mapping, environment map bump mapping, and the highly popular dot product bump mapping (normal mapping). For a more detailed description of these techniques look in [Möller02].

Bump mapping doesn't take into account the geometric depth of the surface and therefore does not exhibit parallax. Since this technique displays various visual artifacts (described in detail in the next section), several approaches were introduced to simulate the parallax effect for bump mapped geometry. However, existing parallax techniques cannot account for self-occluding geometry or add shadowing effects. Indeed, shadows provide a very important visual cue for surface detail. The main contribution of this article is an advanced technique for simulating the illusion of depth on uneven surfaces without increasing the geometric complexity of rendered objects. This is accomplished by computing a perspective-correct bump map preserving parallax effect using a novel reverse height map tracing technique. We also describe a method for computing self-shadowing effects for self-occluding objects. The resulting approach allows us to simulate pseudo-geometry displacement in the pixel shader instead of modeling geometric details polygonally. This allows us to render surface detail providing a convincing visual impression of depth from varying viewpoints, utilizing the per-pixel capabilities of the latest generation of graphics hardware.

Common Bump Mapping Artifacts

When standard bump mapping techniques are applied per-pixel, the appearance of surface detail is marginally believable. Although they offer a relatively inexpensive way to add surface detail, there are several downsides to these techniques. Common bump mapping techniques lack the ability to represent view-dependent unevenness of detailed surfaces, and therefore fail to represent the motion parallax effect—the apparent displacement of the object due to viewpoint change. In recent years, new approaches for simulating displacement on surfaces have been introduced. [Kaneko01] and [Welsh03] described an approach for parallax mapping for representing surface detail using normal maps, [Wang03] introduced a technique for view-dependent displacement mapping which improved displaying surface and silhouette detail.

Another limitation of bump mapping techniques is the inability to properly model self-shadowing of the bump mapped surface, adding an unrealistic effect to the final look. [Sloan00] described a horizon mapping technique, first introduced in [Max88]. With this approach the height of the shadowing horizon at each point on the bump map for eight cardinal directions is encoded in a series of textures which are used to determine the amount of self-shadowing for a given light position during rendering. A variety of other techniques were introduced for this purpose. Again, the reader may refer to an excellent survey in [Möller02].

A further drawback is the smooth silhouettes of bump mapped objects, not displaying the surface detail on the boundaries (since during bump mapping only the normal used in the lighting equation is perturbed, not the actual geometric surface normal). The approach described in this article does not address this concern; however, several approaches have been implemented to resolve this issue. Displacement mapping is an excellent solution for this problem. It is a technique for tessellating the original surface into a large number of small triangles, vertices of which are displaced by using a supplied height function. However a major disadvantage of this approach is the fact that it isn't available in most consumer hardware at the present time and that displacement mapping requires fairly highly tessellated models in order to achieve satisfactory results, thus negating the polygon-saving effect of bump mapping.

[Wang03] describes a per-pixel technique for self-shadowing view-dependent rendering capable of handling occlusions and correct display of silhouette detail. The precomputed surface description is stored in multiple texture maps (the data is precomputed from a supplied height map). The view-dependent displacement mapping textures approach displays convincing parallax effect by storing the texel relationship from several viewing directions. However, the cost of storing multiple additional texture maps for surface description is prohibitive for most real-time applications; despite the fact that this technique does produce very realistic images at high frame rates.

Parallax Occlusion Mapping

Parallax mapping approximates the correct appearance and lighting of uneven surfaces by modifying the texture coordinate for each pixel [Kaneko01]. The effect of parallax is seen by the viewer when portions of a surface appear to move relative to one another due to a change in the view position. Thus parallax is a very important visual cue for viewing non-flat surfaces. However, creating a complex geometric object to simulate the details of the surface is computationally expensive and thus prohibitive to real-time rendering practices. Instead surface height data can be used for approximating surface appearance using techniques described in this chapter.

The effect of motion parallax for a surface can be computed by applying a height map with additional surface details and offsetting each pixel in the height map using the geometry normal and the eye vector. As you move the geometry away from their original positions using that vector, the parallax is obtained by the fact that the highest points on the height map would move the farthest along that vector and the lower extremes would appear not to be moving at all. To obtain satisfactory results for true perspective simulation, one would need to displace every pixel in the height map using the eye vector and the geometry normal. However, that approach can be fill-limiting and computationally expensive for large height maps in real-time scenarios, although it yields excellent results simulating the parallax effect for the entire surface.

Tangent Space

Before we can introduce a detailed explanation of real-time computation of parallax effect, we must clarify the concept of tangent space since our technique relies on this understanding. Bump mapping works by perturbing the geometric normal or replacing it altogether with a normal vector stored in a texture map. The normal vectors are computed in the tangent space for each vertex. Tangent space is used for orientation of the texture map at each vertex. Since texture coordinates basis vectors $\bar{u}$ and $\bar{v}$ can be oriented arbitrarily at each vertex, a question arises in regard to the correct orientation for applying texture maps. We can use the tangent space basis vectors $\bar{T}$ and $\bar{B}$ to orient $\bar{u}$ and $\bar{v}$ respectively.

From a mathematical standpoint, if we view a polygonal vertex as a point on a differentiable manifold, this vertex has a real vector space that contains all possible directions for passing through the given point tangent to the surface. Although we are operating on 3D objects, the actual surfaces that we are computing lighting equations for are in fact 2D manifolds. For 2D surfaces, the tangent space is planar (generally speaking, tangent space has the same dimensions as the manifold's dimensions). It can be pictured as follows: if the given surface is a sphere, the tangent space at each sphere's vertex is the plane that touches the sphere at that point and is perpendicular to the sphere's normal through that point.

Figure 2.5.1 displaces tangent space basis vectors on a sphere and a torus, where vector T is the tangent vector, $\overline{B}$ is the binormal vector, and $\overline{N}$ is the normal vector for each vertex.

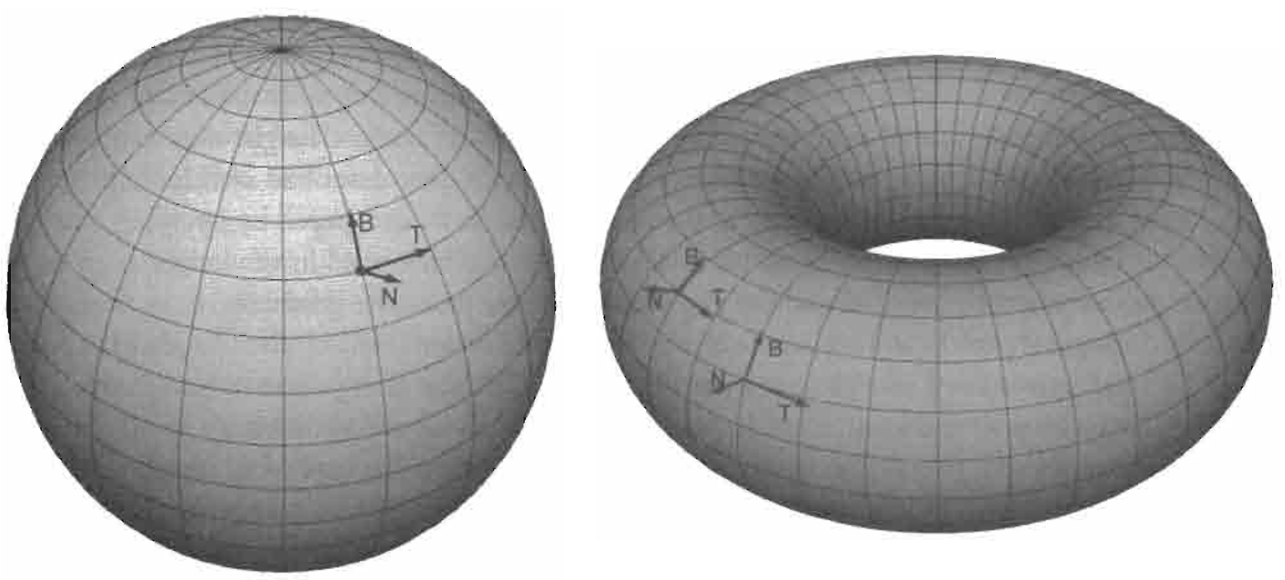

FIGURE 2.5.1 *Tangent space basis vectors on sphere and torus manifolds.*

Given a normal in tangent space, in order to correctly compute the result of the lighting equation for any point on the surface, we need to transform the light vector into tangent space. That's done via a simple matrix multiplication by a tangent space transformation matrix formed by unit tangent, binormal, and normal vectors as follows:

$$\begin{pmatrix} t_x & t_y & t_y & 0 \\ b_x & b_y & b_z & 0 \\ n_x & n_y & n_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

where $(b_x, b_y, b_z$ is the binormal vector, computed by simply taking the cross product $\overline{B} = \overline{N} \times \overline{T}$. Typically the tangent basis vectors are precomputed at preprocessing time. Note that the light vector must be transformed into tangent space separately for each vertex, since for non-planar objects, tangent basis vectors will vary for each vertex.

Basic Parallax Mapping

In order to understand how parallax mapping works, we first visualize a height map representing the geometry of a simple pyramid, mapped onto a flat plane (Figure

2.5.2). If we display this object with plain texture mapping without shifting texture coordinates, the geometry will appear flattened because we are not fully taking the height map into account. If we were to view the actual geometry for the pyramid, the elevated points on the pyramid would correctly move away from the viewer and the points in valleys would move toward the eye. This aids the perspective perception of the geometry in space. In order to achieve this effect, we must displace the texture coordinates for sampling the height map in texture space according to the apparent screen space distortion. Therefore the viewed objects will appear to shift in perspective thus simulating their actual geometric properties.

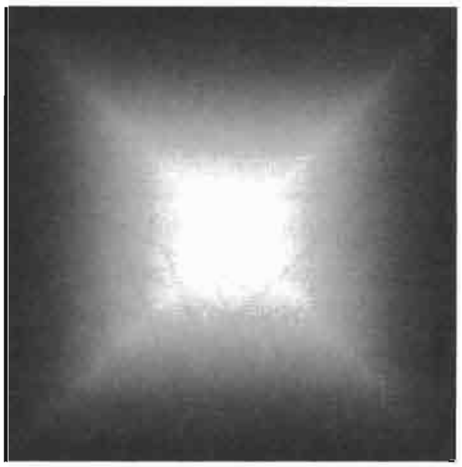

FIGURE 2.5.2 *A height map representing a pyramid.*

Figure 2.5.3 shows how the texture map for this pyramid would distort in texture space in order to appear as if it is shifting in perspective when mapped onto a flat surface.

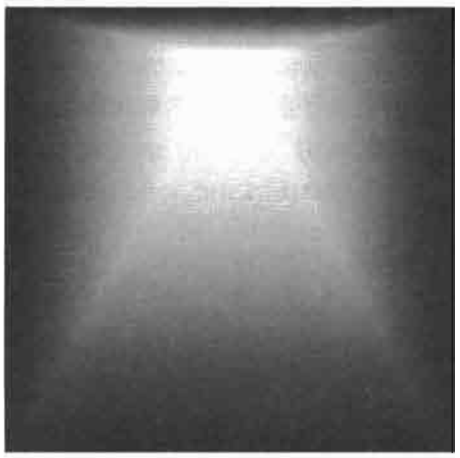

FIGURE 2.5.3 *Perspective distortion for a pyramid when mapped onto a plane.*

One way to create texture distortion to simulate parallax is to use the view vector to shift samples from a height map. This approach samples the height map per-pixel at the original texture coordinates, scales and biases it, and then uses the view vector at that point to shift the texture coordinates. This is described in more detail in [Welsh04]. The vector used to offset the texture coordinates in the direction of the view vector is called the parallax vector. Figure 2.5.4 shows the vectors in tangent space, where $\bar{N}$ is the normal vector at the sampled point, $\bar{T}$ is the tangent vector, $\bar{B}$ is the binormal vector, $\bar{E}$ is the eye (view) vector, and $\bar{P}$ is the desired parallax vector.

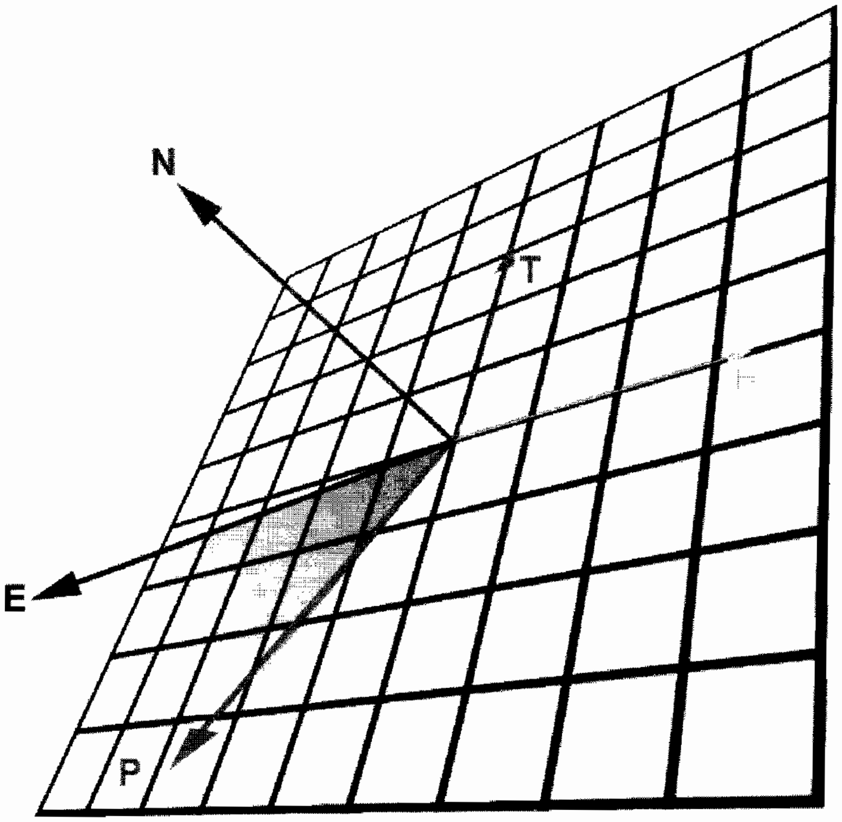

FIGURE 2.5.4 *Tangent space vectors.*

The length of this parallax vector represents the distance that the highest points in the height map for the original sample would displace in texture space. The pixel is then displaced in the height map along the parallax vector by a distance that is a percentage of the parallax vector's length based on the height map values at the starting sample point. In order to get satisfactory results we would need to set appropriate scale and bias based on the actual represented height values in the height map. The resulting shifted texture coordinate can be used to sample the height map approximating the same sample point that we would have if we had actual geometric object representation.

One serious visual artifact with this simple approach is the resulting shimmering of textured pixels when viewed at grazing angles when the view vector is nearly perpendicular to the surface normal. This happens because at these viewing angles, texture coordinate displacement values tend to be large, and for objects with varying per-pixel heights, this results in low coherency for height map samples, thus producing a strong shimmering effect. In order to reduce this shimmering, [Welsh04] introduces an offset limiting concept, whereby we would limit the parallax displacement for each pixel so that it doesn't exceed the original height sampled from the height map at the starting texture coordinate. Although this technique reduces shimmering, since it's simply a heuristic, it doesn't perform in a reasonable fashion for many complicated height maps.

Parallax Occlusion Mapping

In order to improve the approach for computing parallax, we describe a novel technique using the reverse of the above process to efficiently achieve parallax. The core idea is to trace the pixel being currently rendered back in the height map to determine which texel in the height map would yield the rendered pixel location if in fact we would have been using actual displaced geometry.

To determine a new location for the perspective-correct pixel, we use the parallax vector. In order to compute the parallax vector, we construct a plane between the view vector and the geometric normal from the normal map of the surface. The parallax vector lies on the intersection of this plane with the tangent plane at the sample point. To determine the length of the parallax vector we can use the world space apparent height for the displaced geometry and the angle between the view vector and the parallax vector. The desired displaced pixel will lie along the profile of the parallax vector. We will next sample the height map along the direction of the reversed parallax vector. The accuracy of this technique depends on the number of samples that we will take from the supplied height map. By plotting the resulting samples we construct a profile of the displaced geometry along the parallax vector. We create a line by tracing from the maximum displacement at the end of the reversed parallax vector to the original starting point. The intersection point of that line will yield the perspective-correct displacement location that we set out to find.

This technique also allows for the simulation of geometric occlusion and self-shadowing effects. By using the light vector instead of the view vector in the above algorithm we can create proper self-shadowing effect for perspective-shifted bump maps to further increase the illusion of depth and surface detail.

For this method to work properly, it would require moving every pixel in the texture map [and filling in gaps between them], and the result would have to be completely recalculated for every pixel on the screen. To apply this in any useable application the solution is to instead back trace on the reversed parallax vector, sampling as many points as possible, and solve for the point which should displace forward to the point we started from.

Determining Perspective-Correct Parallax Displacement Offset

Determining an accurate parallax displacement vector is at the heart of this approach. Although we are going to be computing this displacement value per-pixel, we can improve the efficiency of the algorithm by computing the initial parallax displacement vector in the vertex shader, and then by simply using hardware interpolators to yield the resulting per-pixel starting parallax displacement vector. In order to compute this displacement vector, we need to first calculate the view, tangent, binormal and normal vector for the point. Since we are computing an offset vector for sampling the height map in tangent space, all of these vectors must first be transformed into tangent space and then normalized.

To simulate a height map for convex surfaces extending outward from the flat geometry, we compute the parallax displacement vector using dot products of the view, tangent, and binormal vectors. Figure 2.5.5 shows the vectors where $\bar{N}$ is the surface normal, $\bar{T}$ is the tangent vector, $\bar{B}$ is the binormal, $\bar{E}$ is the view (eye) vector, and $\bar{P}$ is the reversed parallax displacement vector.

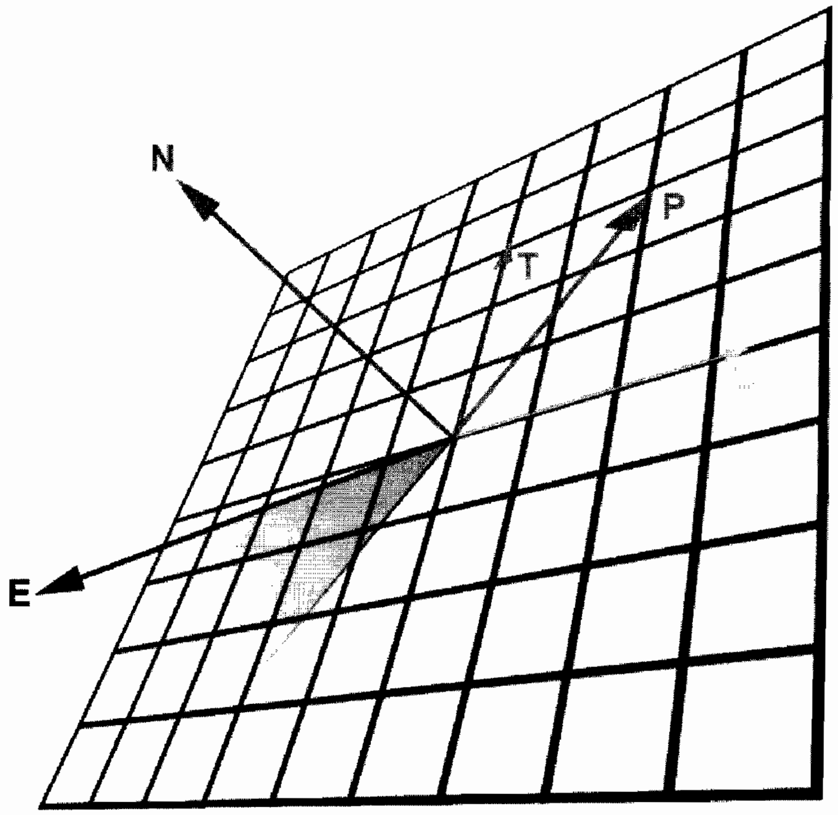

FIGURE 2.5.5 *Tangent space reversed parallax vector.*

To determine the direction of the parallax vector for texture displacement, we compute it as:

$$P_x = \bar{E} \cdot \bar{T}$$
$$P_y = \bar{E} \cdot \bar{B} \tag{2.5.1}$$

This will give us the correct direction for the parallax vector. However, if we simply use the length of this vector as the actual displacement amount, we would achieve incorrect results. Although this parallax vector would display proper direction for shifting of the texture coordinate, it would miscalculate the displacement amount, resulting in an excessive parallax shift when the surface is angled towards the camera, and lack of parallax shift as the surface approaches angles perpendicular to the camera plane.

Therefore we must compute a more accurate parallax vector. In order to do that, we introduce a parallax binormal vector, $\bar{P}_b$, which is the cross product between the normal vector and the view vector (see Figure 2.5.6 for visualization of these vectors in tangent space for the sample point, where $\bar{N}$ is the surface normal, $\bar{T}$ is the tangent vector, $\bar{B}$ is the binormal, $\bar{E}$ is the eye (view) vector, $\bar{P}_b$ is the parallax binormal vector, and $\bar{P}$ is the parallax displacement vector):

$$\bar{P}_b = \bar{N} \times \bar{E} \tag{2.5.2}$$

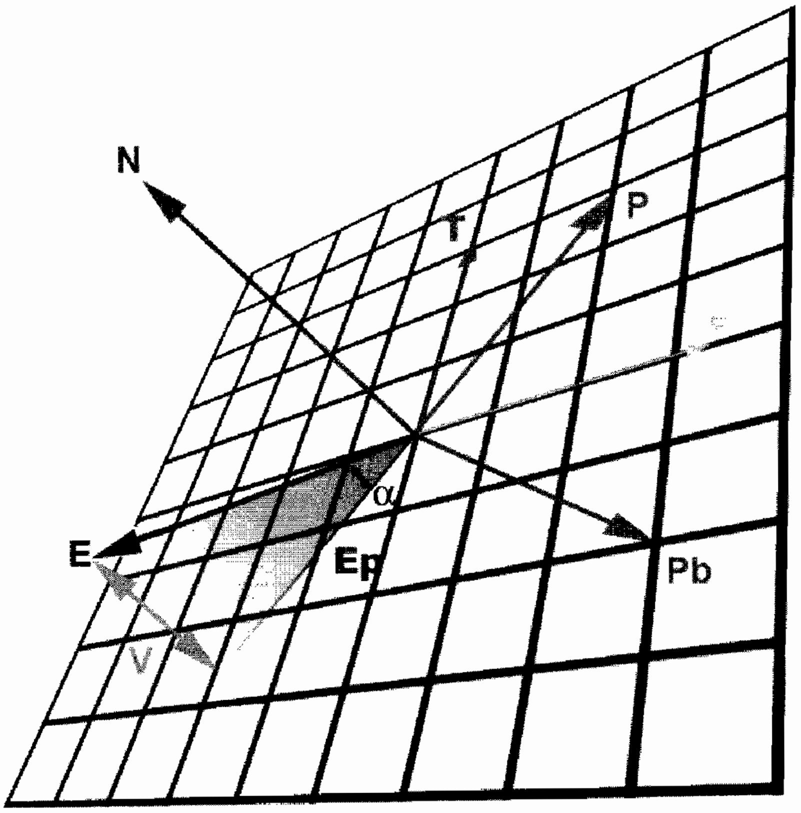

FIGURE 2.5.6 *Vectors used for parallax displacement amount computation.*

Then, using this parallax binormal vector, we compute the new parallax displacement vector as follows:

$$\bar{P} = \bar{N} \times \bar{P}_b \qquad (2.5.3)$$

Next, we need to determine the length of the parallax displacement vector. This is crucial for the correct approximation of the desired displaced height for the sample pixel since the displacement amount will determine the apparent height for the given point and thus will be the controlling factor for a good parallax effect. If we were to use the input height map to actually displace geometry, the amount of height displacement would be equal to the quantity *V* that we see in Figure 2.5.6, which is the apparent vertical scale for parallax. Notice that this value is highly dependent on the content created by the artist. To solve for the amount of parallax displacement, i.e., parallax vector length, we should look at the right angle triangle formed by vectors $\bar{V}$, $\bar{E}$, and $\bar{E}_p$ (the view vector projected onto the tangent plane). The length of the parallax displacement amount will be equal to the length of $\bar{E}_p$. The angle between $\bar{E}$ and $\bar{E}_p$, α, can be computed as follows:

$$\alpha = \text{acos}\,(\bar{E} \cdot \bar{P}) \qquad (2.5.4)$$

where $\bar{P}$ is the parallax displacement vector computed in Equation 2.5.4. Note that $\bar{P}$ should be normalized prior to computing (Equation 2.5.4).

Then using the result obtained from Equation 2.5.4, we can compute the length of the parallax displacement as:

$$l_p = \frac{1}{\tan(\alpha)} * |\bar{V}| \qquad (2.5.5)$$

Create a 2D vector from the *x* and *y* displacement values, normalize its length, and multiply that by the parallax length and we now have correct *x* and *y* displacement values in texture space. Our goal is to determine which points in the height map will displace to the sample point currently being rendered. In order to calculate these sample points, we need to sample the height map in the reverse direction of the apparent parallax vector. Therefore we negate the computed parallax displacement vector for resulting computations.

After some simplifications, we obtain the following vertex shader for computing parallax displacement direction and amount:

Vertex shader for computing parallax displacement vector (vs_2_0):

```
float4x4 matViewInverse;
float4x4 matViewProjection;
float4x4 matView;

float fBaseScale;
float fHeightScale;
float fPerspectiveBias;

struct VS_OUTPUT
{
   float4 Position    : POSITION;    // Transformed position
   float2 TexCoord    : TEXCOORD0;   // Initial texture coordinate
   float2 vParallaxXY : TEXCOORD1;   // Parallax displacement vector
};

VS_OUTPUT vs_main(
   float4 inPosition: POSITION,
   float2 inTexCoord: TEXCOORD0,
   float3 inNormal  : NORMAL,
   float3 inBinormal: BINORMAL,
   float3 inTangent : TANGENT)
{
   VS_OUTPUT Out = (VS_OUTPUT) 0;

   // Transform and output vertex position:
   Out.Position = mul( matViewProjection, inPosition );

   // Propagate texture coordinates:
   Out.TexCoord = inTexCoord;

   // Compute view vector (in view space):
   float3 vView = -mul( matView, inPosition );

   // Compute tangent space basis vector (in view space):
   float3 vTangent  =  mul( matView, inTangent  );
   float3 vBinormal =  mul( matView, inBinormal );
   float3 vNormal   =  mul( matView, inNormal   );

   // Normalize the view vector in tangent space:
   float3 vViewTS = mul( float3x3( vTangent, vBinormal, vNormal),
                      normalize( vView ) );

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

   // Compute the length of parallax displacement vector
   // i.e. the amount of parallax displacement.
   float fParallaxLength= -sqrt(1 - vViewTS.z * vViewTS.z ) /
vViewTS.z;

   // Compute parallax bias parameter which allows
   // the artists control over the parallax perspective
   // bias via an artist editable parameter 'fPerspectiveBias'
   // Use view vector length in tangent space as an offset
   // limiting technique as well:
   float fParallaxBias = fPerspectiveBias + (1 - fPerspectiveBias) *
                      (2 * vViewTS.z - 1 );

   // Compute the actual reverse parallax displacement vector:
   Out.vParallaxXY = -vParallaxDirection * fParallaxLength *
                   fParallaxBias * fHeightScale;

   return Out;
}
```

Reverse Height Map Tracing

In order to understand the process for reverse height map tracing, let's first imagine that we are trying to determine parallax displacement for the original sample point P_s in Figure 2.5.7. The green curve in this figure represents the actual profile of the input height map. Note that if we were to directly sample the height map at P_s, we would see the height profile for point 0 along the green line. This point represents our original, non-displaced texture coordinate for a given pixel. However, for a given viewing direction, when simulating accurate parallax, the viewer actually should be seeing point P_d instead. How do we arrive at this point P_d given a reversed parallax vector $\bar{P}$?

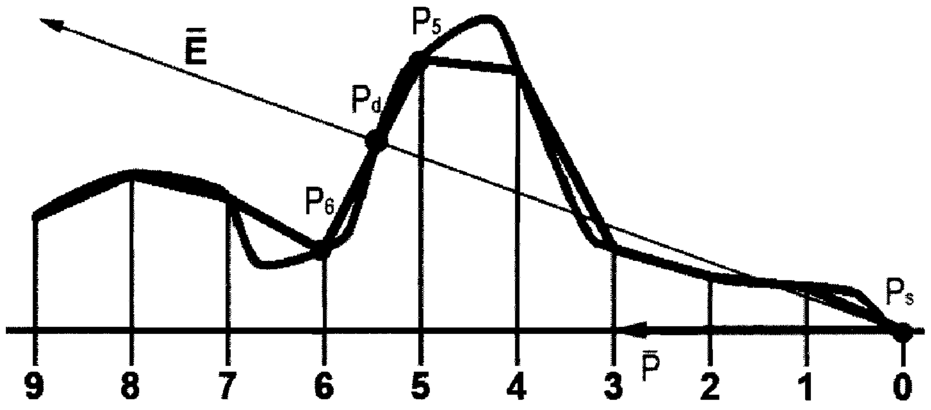

FIGURE 2.5.7 *The process for reverse height map tracing.*

If we were to trace along the reversed parallax vector $\bar{P}$ and discretely sample the height map along the intervals 0–9, the resulting discrete height map samples will yield a profile drawn as a thick blue line over the green profile of actual continuous height map values.

In order to compute the location of point P_d, we should start at interval 9 and trace along the parallax vector toward the starting point, computing the height map sample profile. The first intersection between the sampled height map profile and the view vector will yield the desired height value that the viewer would see if the height map was actually used to displace the geometry.

In order to compute the intersection point P_d, we test each sample height value for each interval to see if it's above the height value of the view vector at that point. The first time when the sampled height value is higher than the view vector height (as projected onto the tangent plane) will determine the segment of the height profile that will contain the desired displacement height. The intersection point P_d will be located between the found sample point and its previous interval. To yield the actual value, we solve the line-line intersection equation between the view vector $\bar{E}$ and the height sample profile between the points that contain P_d. In the example shown in

Figure 2.5.7, the desired intersection point P_d falls in the interval between samples 5 and 6. The line formed by the sampled height map values P_5 and P_6 is intersected with the view vector to yield the actual value for P_d.

Pixel shader for computing parallax displaced coordinate (ps_2_0):

```
sampler tNormalMap;

float4 ps_main( float2 inTexCoord: TEXCOORD0,
                float2 vParallax : TEXCOORD1 ): COLOR0
{
   // Compute height map profile for 8 samples:
   // Note that the height map is contained in the alpha channel
   // of the input normal map:
   float h0 = tex2D( tNormalMap, inTexCoord - vParallax * 1.000 ).w;
   float h1 = tex2D( tNormalMap, inTexCoord - vParallax * 0.875 ).w;
   float h2 = tex2D( tNormalMap, inTexCoord - vParallax * 0.750 ).w;
   float h3 = tex2D( tNormalMap, inTexCoord - vParallax * 0.625 ).w;
   float h4 = tex2D( tNormalMap, inTexCoord - vParallax * 0.500 ).w;
   float h5 = tex2D( tNormalMap, inTexCoord - vParallax * 0.375 ).w;
   float h6 = tex2D( tNormalMap, inTexCoord - vParallax * 0.250 ).w;
   float h7 = tex2D( tNormalMap, inTexCoord - vParallax * 0.125 ).w;

   // Determine the section in the height profile that contains
   // a sample that is higher than the view vector. This determines
   // the correct displacement point:
   float x,  y;
   float xh, yh;

   if      ( h7 > 0.875 ) { x = 0.875; y = 1.000; xh = h7; yh = h7; }
   else if ( h6 > 0.750 ) { x = 0.750; y = 0.875; xh = h6; yh = h7; }
   else if ( h5 > 0.625 ) { x = 0.625; y = 0.750; xh = h5; yh = h6; }
   else if ( h4 > 0.500 ) { x = 0.500; y = 0.625; xh = h4; yh = h5; }
   else if ( h3 > 0.375 ) { x = 0.375; y = 0.500; xh = h3; yh = h4; }
   else if ( h2 > 0.250 ) { x = 0.250; y = 0.375; xh = h2; yh = h3; }
   else if ( h1 > 0.125 ) { x = 0.125; y = 0.250; xh = h1; yh = h2; }
   else                   { x = 0.000; y = 0.125; xh = h0; yh = h1; }

   // Compute the intersection between the view vector and the high-
est
   // sample point in the height profile:
   float fParallaxEffect = ( x * (y - yh) - y * (x - xh) ) /
                           (( y - yh ) - ( x - xh ) );

   return float4( vParallax * (1 - fParallaxEffect), 0.0, 1.0 );
}
```

Note that the accuracy of this calculation depends on the number of samples taken for the height map profile. In reality this factor is gated by the number of available pixel shader texture fetches. If one desires to improve the accuracy of resulting parallax occlusion mapping computation, separate sections of height profiles can be computed with greater accuracy in individual passes and composited together in the final pass, thus resulting in a multipass approach. Another way to improve the accuracy is to use more advanced shader models available on the latest graphics hardware, for example, pixel shaders ps_2_b, which contain a greater number of ALU instructions and texture fetches. The desired accuracy depends greatly on the quality and intent of the art content developed for this effect.

Lighting for Pseudo-Displacement Mapping Effect

The height value of the intersection point P_d is assumed to be a value in the 0 to 1 range. Using this value and the original parallax displacement direction and amount, we can generate a more correct perspective displacement value, d_{offset}. Using the original texture coordinates for a given pixel and displacing them by the new d_{offset} amount, we compute an accurate position in texture space for representing true parallax shift of the apparent texture by the height map as seen by the viewer. We can use the new displaced texture coordinate for all subsequent queries for applied texture maps, such as a normal map, diffuse, specular, detail maps, etc. This results in the surface of the object having the distinct appearance of displacement mapping using the input height map (see Figure 2.5.8). Thus we've effectively achieved geometric pseudo-displacement at the pixel shader level.

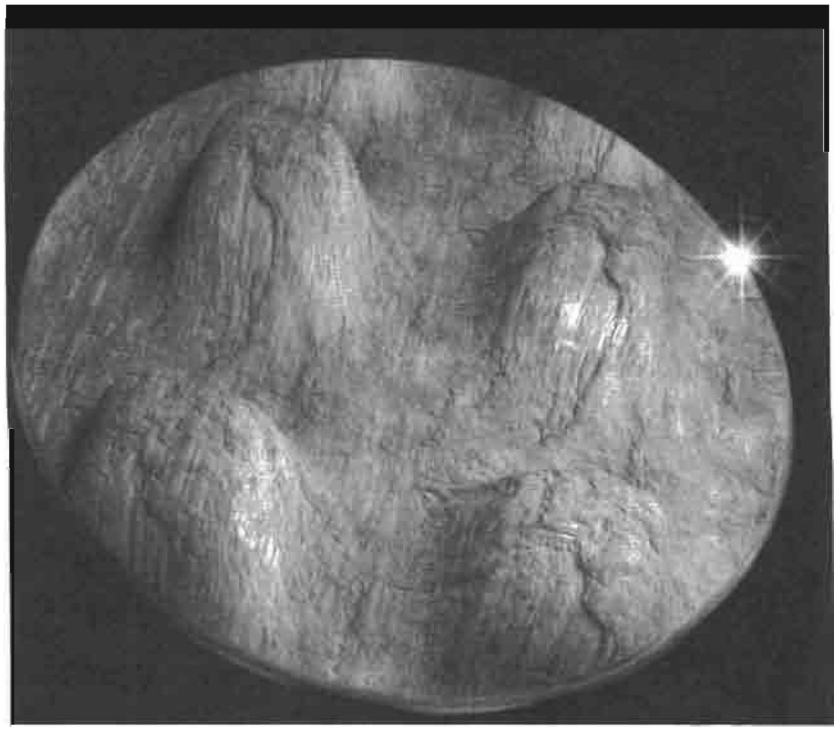

FIGURE 2.5.8 *Result of applying lighting with parallax occlusion mapping.*

Since reverse height map tracing in the pixel shader 2.0 model typically consumes all available texture fetch instructions, we compute the lighting effects for the surface in a subsequent pass. However, in pixel shader 2.b we can compute both parallax displacement and the lighting for the surface in the same pixel shader, which performs very efficiently on the latest generation of graphics hardware. The vertex shader and the pixel shader for the lighting pass can be found in the RenderMonkey parallax occlusion workspace on the CD-ROM accompanying this book.

ON THE CD

Self-Occlusion and Shadowing Computation

An additional advantage of the parallax occlusion mapping technique lies in the fact that it can be used to simulate self-occlusion and shadowing effects. The height map can in fact cast shadows on itself. This is accomplished by substituting the light vector for the view vector when computing the intersection of the height profile to determine the correct displaced texel position during the reverse height mapping step. We can reuse the parallax displacement vector computed in an earlier pass. We do not solve for the first intersection between the light vector and the height profile. Instead we simply test whether any point in the sampled height profile is above the light vector for the sample point. If that is the case, we then know that the height map geometry for the sampled profile is in fact blocking the light source for the sample point, and therefore it will be in shadow. Figure 2.5.9 shows shadows computed as a result of this test.

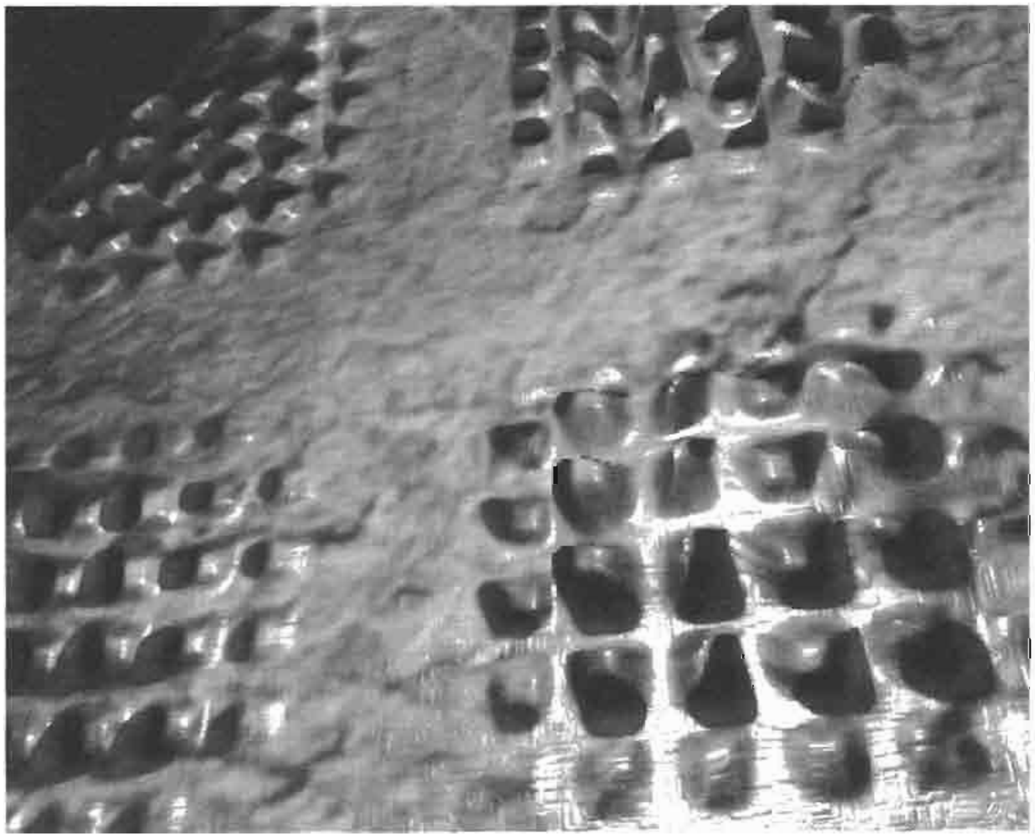

FIGURE 2.5.9 *Self-Occlusion shadowing effects with parallax occlusion mapping.*

Notice that the shadows are correctly aligned with the occluded displaced texture. In order to achieve this effect, the height map values used to create the sampled height map profile for this test must use the displaced texture coordinates computed during the parallax occlusion part of the algorithm. This way the self-occlusion shadows correctly line up with pseudo-displaced geometry. If we were to use the original input texture coordinate during computations of the shadows, they would appear floating on the surface of the object (as shown in Figure 2.5.10) since they would actually be computed for initial sample points, rather than for displaced texture coordinates.

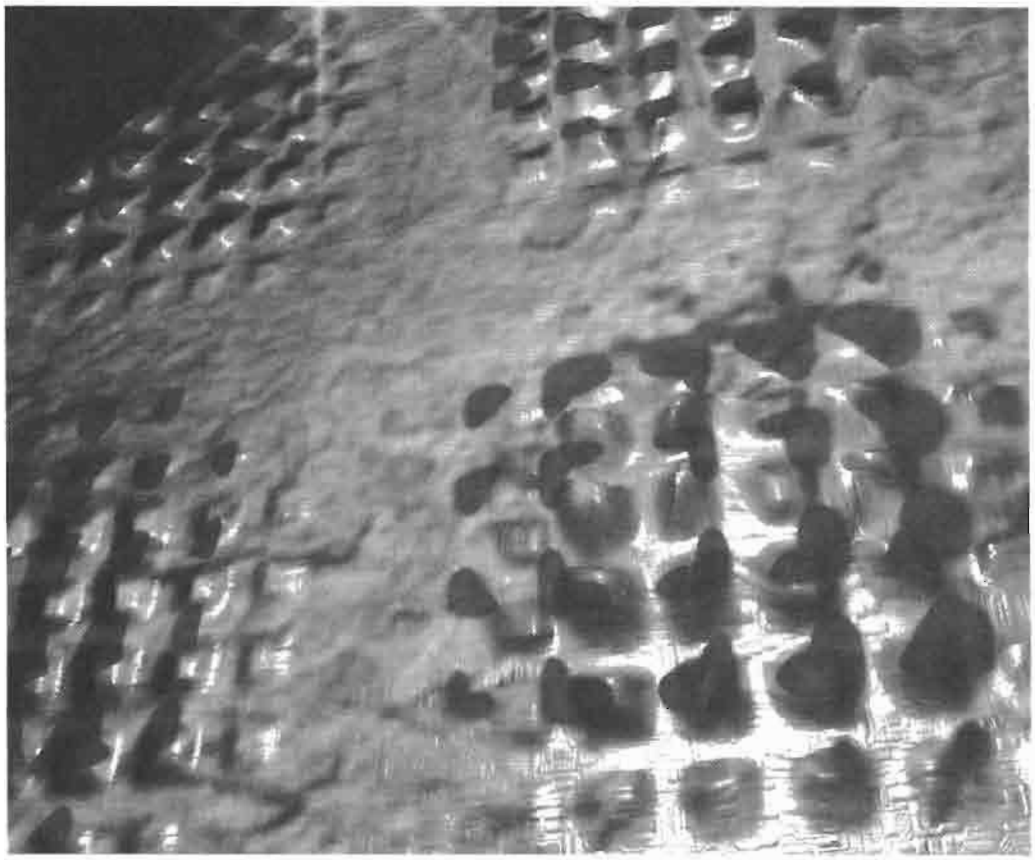

FIGURE 2.5.10 *Incorrect shadow resulting due to non-shifted texture sample point.*

Computing shadows with parallax occlusion mapping is a straightforward technique to implement and yields very convincing self-shadowing effects that can be otherwise very expensive to compute. This technique also doesn't require much additional memory cost, since for multipass approaches we can output the shadow map in the alpha channel of the parallax occlusion map computed during the parallax displacement part of the algorithm. If we are operating in shader model 2.b then we can simply use longer shaders to compute a shadowing term before calculating the lighting equation result.

Limitations of the Parallax Occlusion Mapping Technique

Although parallax occlusion mapping is a very powerful and flexible technique for computing real-time lighting in detailed objects, we must make the readers aware of its limitations. One of the key limitations of this technique is that any peak in the height map profile which falls between the height map sample points will be missed. If we look at Figure 2.5.11, we will see that peaks in sample regions 9–8, 8–7, 7–6, 4–3, and 3–2 will be missed by the sampled height map profile due to insufficient sampling frequency.

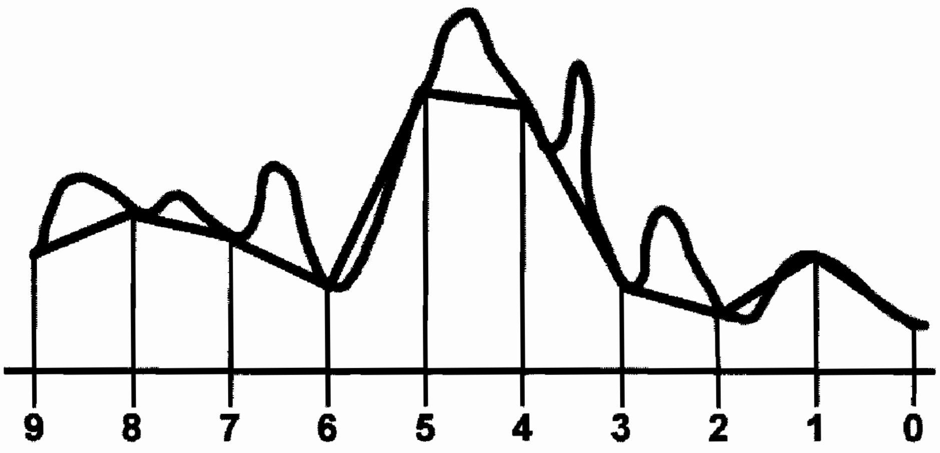

FIGURE 2.5.11 *Missed height details resulting due to very high-frequency height map profile.*

Another obvious downside of this approach is the fact that it will not create apparent object detail along the boundaries of the displayed objects (unlike displacement mapping technique, which exhibits correct object silhouettes since it actually modifies geometry). This limitation is common to all bump mapping techniques since they rely on perturbed surface normal used for lighting computations, not by modifying geometry directly.

When parallax-mapped objects are viewed at angles nearly perpendicular to the surface normal, the parallax displacement amount approaches infinity. Therefore entire features in the height map may be missed completely. This results in severely flattened objects when viewed at grazing angles. This is particularly visible when displacing planar objects. Figure 2.5.12(a) shows a nearly planar disc with parallax—all of the bumps display correct profile along the surface of the disc. However, if we were to rotate this disc so that the view vector is nearly perpendicular to the surface normal, we would see that most of the height details have disappeared from the surface (Figure 2.5.12(b)) and the height map would appear distorted and flat.

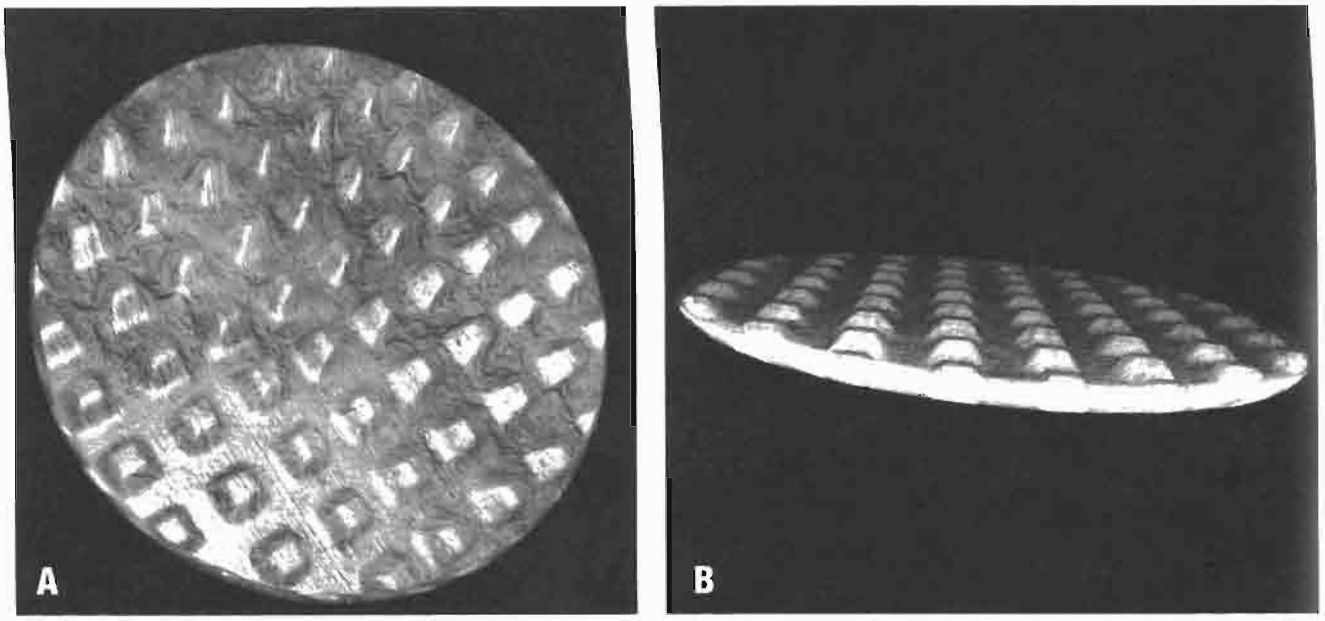

FIGURE 2.5.12 *Steep viewing angles of artifacts of parallax mapping. (a) Correct perspective parallax display (b) Flattened height map due to grazing view angles.*

This artifact is common to all parallax approaches. One solution to this problem is to introduce an additional parameter during the computation of the parallax displacement amount in the vertex shader. The parameter can reduce the displacement amount when the angle between the view vector and the surface normal approaches 90°. This is similar to the offset limiting approach described in [Welsh04]. However, we use a more accurate observation to constraint the parallax distortion amount as a function of the dot product between the surface normal and the view vector. To allow the artist a level of control for finer parallax effect, we can also further constraint the parallax displacement amount by raising the above dot product to the power of some value *n*, where *n* is the artist-controlled parameter for visually adjusting the optimal balance for oblique angle distortion due to separation between nearby parallax levels in texture space.

Non-planar objects do not exhibit this artifact as strongly, so another solution can be to simply use slightly convex geometry instead of flat surfaces (see Figure 2.5.13).

Art Content Suggestions for Parallax Occlusion Mapping

Parallax occlusion mapping technique is a very efficient and compelling technique for simulating surface details. However, as with many other bump mapping techniques, its quality depends strongly on the quality of its art content. Empirically, we found that low-frequency height map textures yield much more coherent parallax due to the limitations described in an earlier section of this chapter. Certain types of high-

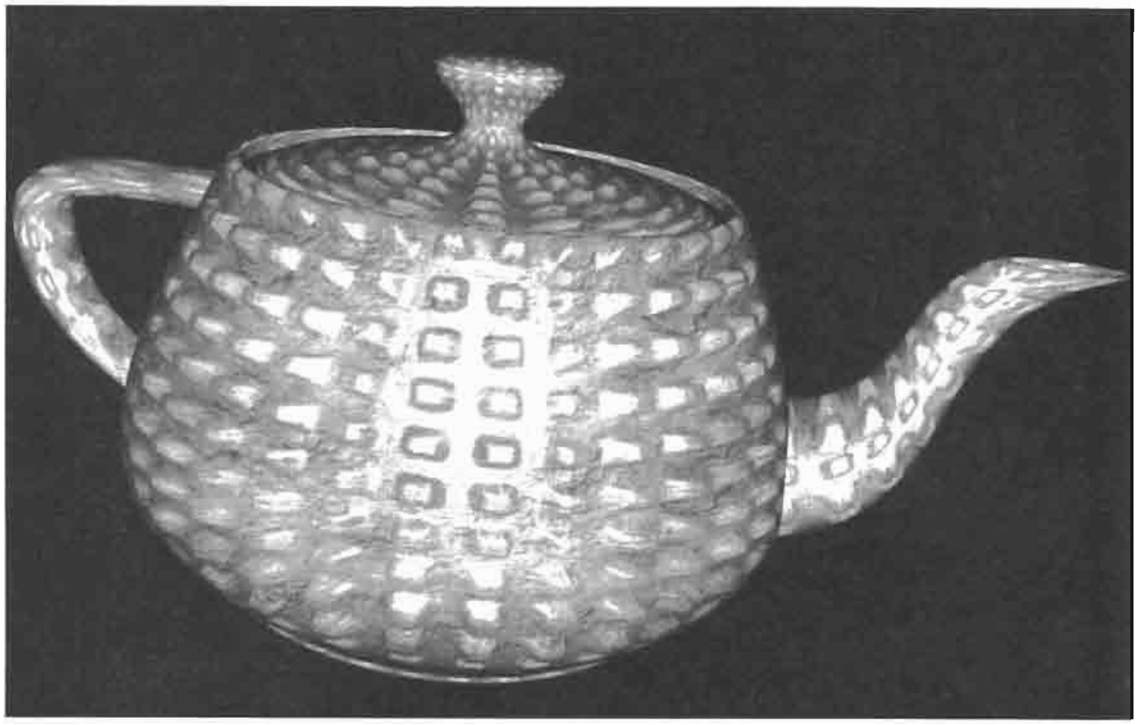

FIGURE 2.5.13 *Non-planar geometry with parallax occlusion mapping.*

frequency height maps do perform adequately, such as, for example, a pyramid height map with straight edges (see Figure 2.5.2). Generally speaking, however, we recommend avoiding noisy or extremely detailed height maps since that will likely result in shimmering pixels. High-frequency changes, such as very tall thin spikes in the height map, present the biggest problem during parallax simulation using this technique.

For underlying geometry we recommend using planar, spherical, or cylindrical geometry since its smooth rate of change along the surface ensures a high level of coherency for parallax displacement computation. Complex concave geometry yields very poor results and in particular will display visually distracting artifacts in the valleys. Of course if one already has complex concave geometry, this approach simply may not be as necessary. Note that also, if the base texture map has baked-in lighting information, then if there is a need to save pixel shader instructions or avoid the multipass approach, we can just compute the parallax displacement offset and use it to offset the base texture map for a simple, yet convincing effect without additional lighting or shadowing computations. This particular case is very useful for highly-detailed base maps used with simple geometry and low-frequency height maps.

Sample Effect

ON THE CD We have provided a RenderMonkey workspace named "Parallax Occlusion Mapping.rfx" demonstrating our technique on the CD-ROM accompanying this book. This requires RenderMonkey 1.5 or higher.

Conclusion

Parallax occlusion mapping can be used effectively to generate an illusion of very detailed geometry exhibiting correct motion parallax as well as producing very convincing self-shadowing effects. It takes advantage of the per-pixel capabilities of the latest graphics hardware to compute all necessary quantities using the programmable pipeline. We hope to see more games implementing compelling scenes using this technique.

References

[Blinn78] Blinn, James F., "Simulation of Wrinkled Surfaces," *Computer Graphics (Siggraph '78 Proceedings)*, August 1978.

[Möller02] Akenine-Möller,T., Haines, E., "Real-Time Rendering," 2nd Edition, *A.K. Peters*, July 2002.

[Kaneko01] Kaneko, T., Takahei, T., Inami, M., Kawakami, N., Yanagida, Y., Maeda, T., Tachi, S., "Detailed Shape Representation with Parallax Mapping," *ICAT,* 2001.

[Sloan00] Sloan, P-P. J., Cohen, M. F, "Interactive Horizon Mapping," *11th Eurographics Workshop on Rendering*, June 2000.

[Max88] Max, N. L., "Horizon Mapping: Shadows for Bump-mapped Surfaces," *The Visual Computer 4,* 2 (July 1988).

[Heidrich00] Heidrich, W., Daubert, K., Kautz, J., Seidel, H-P., "Illuminating Micro Geometry Based on Precomputed Visibility," *ACM Transactions on Graphics (Siggraph 2000 Proceedings)*, July 2000.

[Welsh04] T. Welsh, "Parallax Mapping with Offset Limiting: A Per Pixel Approximation of Uneven Surfaces," 2004.