Parallax Occlusion in Direct3D 11

Frank Luna
February 11, 2012
www.d3dcoder.net

Introduction to 3D Game Programming with DirectX 11 has a chapter explaining normal mapping and displacement mapping (implemented with hardware tessellation). We saw that we could improve upon normal mapping by tessellating the geometry and displacing it with a heightmap to geometrically model the bumps and crevices of the surface. However, an application may be required to support DirectX 10 for systems that do not have DirectX 11 capable hardware; therefore, we need a fallback technique for when we do not have access to hardware tessellation. Parallax Occlusion Mapping is a technique that gives results similar to tessellation displacement mapping, and can be implemented in DirectX 10; see Figure 1 for a comparison of techniques.

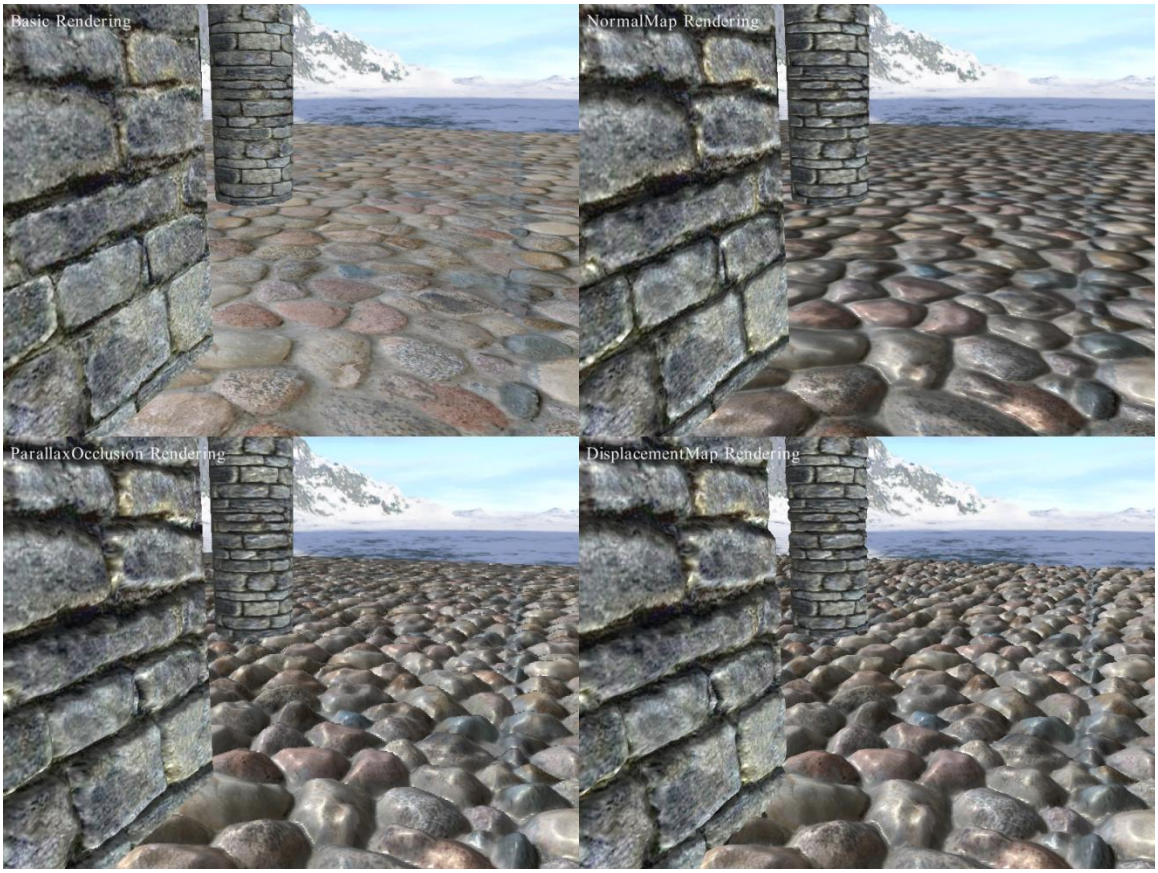

Figure 1: Parallax Occlusion Mapping is a pixel technique that does not require a high triangle count, but gives results comparable to tessellation based displacement mapping.

§1 Conceptual Overview

The basic idea is illustrated in Figure 2. If the surface we are modeling is not truly planar, but defined by a heightmap $h(u, v)$, then the point the eye sees is not the point **p** on the polygon plane, but instead the point **r** where the view ray **v** intersects the heightmap. Therefore, when lighting and texturing the polygon point **p** we should not use the texture coordinates and normal at **p**, but instead use the texture coordinates and normal associated with the intersection point **r**. If we are using normal mapping, then we just need the texture coordinates at **r** from which we can sample the normal map to get the normal vector at **r**. Thus, the crux of the algorithm is to find the *parallax offset vector* **u** that tells us how much to offset the texture coordinates at **p** to obtain the texture coordinates at **r**. As we will see in later sections, this is accomplished by tracing the view ray through the heightmap.

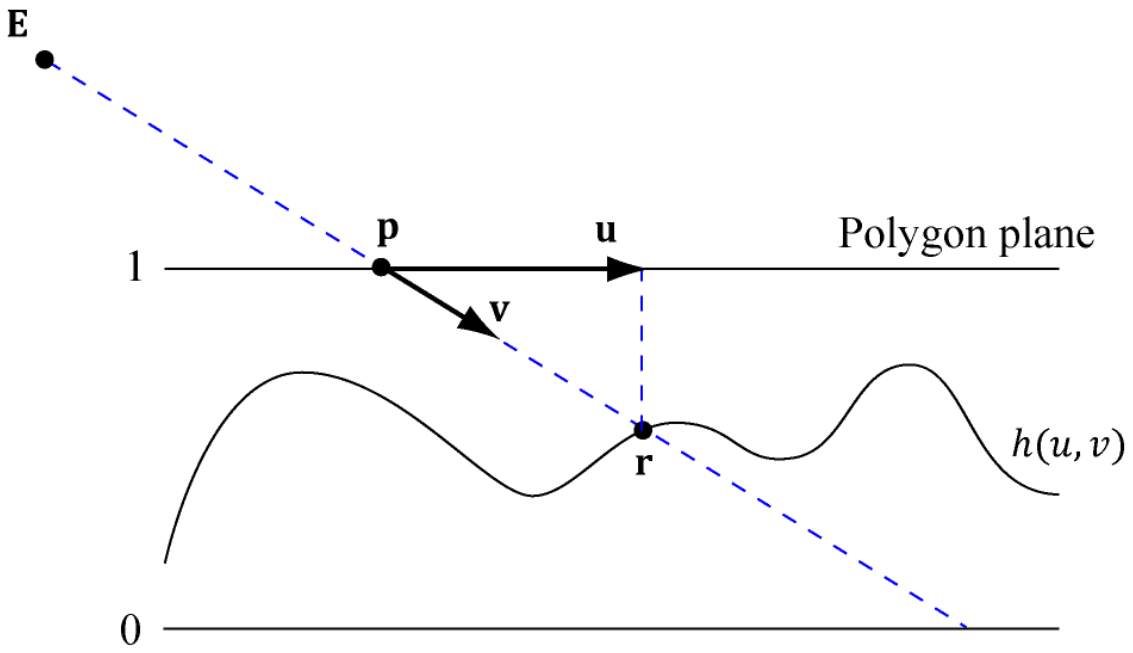

Figure 2: The point the eye sees is not the point p on the polygon plane, but instead the point r where the view ray v intersects the heightmap. The *parallax offset vector* u that tells us how much to offset the texture coordinates at p to obtain the texture coordinates at r.

§2 Computing the Max Parallax Offset Vector

We need an upper limit on how large the parallax offset vector **u** could possibly be. As with our tessellation displacement mapping demo, we use the convention that a value $h = 1$ coincides with the polygon surface, and $h = 0$ represents the lowest height (most inward displacement). (Recall that the heightmap values are given in a normalized range [0, 1], which are then scaled by H to world coordinate values.) Figure 3 shows that for a view ray with starting position at the polygon surface with $h = 1$, it must intersect the heightmap by the time the ray reaches $h = 0$. Assuming the view vector **v** is normalized, there exists a t such that:

$$\mathbf{q} - \mathbf{p} = t\mathbf{v}$$
$$q_z - p_z = tv_z$$
$$t = \frac{q_z - p_z}{v_z}$$

In normalized height coordinates, the difference $q_z - p_z = -1$, but scaled in world coordinates, the difference is $-H$. Therefore,

$$t = \frac{-H}{v_z}$$

So, the view ray $\mathbf{v}$ reaching a height H yields the maximum parallax offset:

$$\mathbf{u}_{\max} = \frac{-H}{v_z}\,\mathbf{v}_{xy}$$

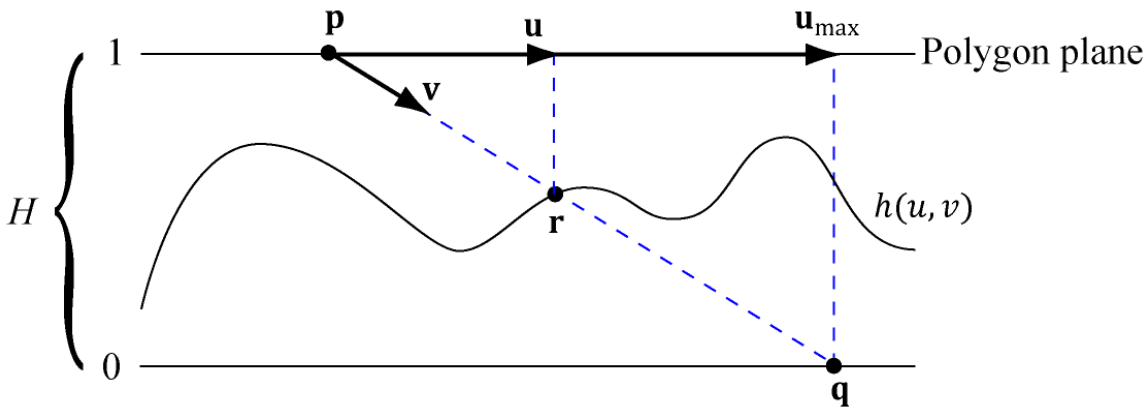

Figure 3: The maximum parallax offset vector corresponds to the point q where the view ray pierces the lowest possible height $h = 0$.

§3 Tex-Coord per World Unit

So far we have made the assumption that we can use $\mathbf{u}$ as a texture coordinate offset. However, if we rotate the normalized view ray from world space to tangent space, the view ray $\mathbf{v}$ scale is still in the "world scale." Moreover, in §2 we computed the intersection of $\mathbf{v}$ with the world scaled height H. Therefore, $\mathbf{u}_{\max}$ is also in the "world scale" and so is $\mathbf{u} = t\mathbf{u}_{\max}$. But to use $\mathbf{u}$ as a texture coordinate offset, we need it in the "texture scale." So we need to convert units:

$$\mathbf{u}' = \mathbf{u} \cdot \frac{ts}{ws}$$

That is to say, we need to know how many world space units (ws) equals one texture space unit. Suppose we have a 20×20 grid in world units, and we map a texture onto the grid with no tiling. Then 1 texture space unit equals 20 world space units and

$$\frac{ts}{ws} = \frac{1}{20}$$

As a second example, suppose we have a 20×20 grid in world units, and we map a texture onto the grid with 5×5 tiling. Then 5 texture space unit equals 20 world space units or 1 texture space unit equals 4 world space units and

$$\frac{ts}{ws} = \frac{texRepeat}{GridSize} = \frac{5}{20} = \frac{1}{4}$$

So to get everything in texture space coordinates so that we can use $\mathbf{u}$ as a texture coordinate offset, we need to revise our equation:

$$\mathbf{u}_{\max} = \frac{-H}{v_z}\,\mathbf{v}_{xy}\,\frac{ts}{ws}$$

In code, $\mathbf{u}_{\max}$ needs to be computed per-pixel. It is unnecessary to perform the calculation $H \cdot \frac{ts}{ws}$ per-pixel. So we can define:

$$\bar{H} = H \cdot \frac{ts}{ws}$$

compute $\bar{H}$ in the application code, and set it as a constant buffer property. Then our pixel-shader code can continue to use the original formula:

$$\mathbf{u}_{\max} = \frac{-\bar{H}}{v_z}$$

In practice, it might not be easy to determine $\frac{ts}{ws}$. An easier way would be to just expose a slider to an artist and let them tweak $H \cdot \frac{ts}{ws}$ interactively.

§4 Intersecting the Heightmap

To determine the point at which the ray $\mathbf{r}(t) = \mathbf{p} + t\mathbf{v}$ intersects the heightmap, we linearly march along the ray with uniform step size and sample the heightmap along the ray (see Figure 4). When the sampled heightmap value h_i is greater than the ray's height value z_i (in tangent space the z-axis is "up"), it means we have crossed the heightmap.

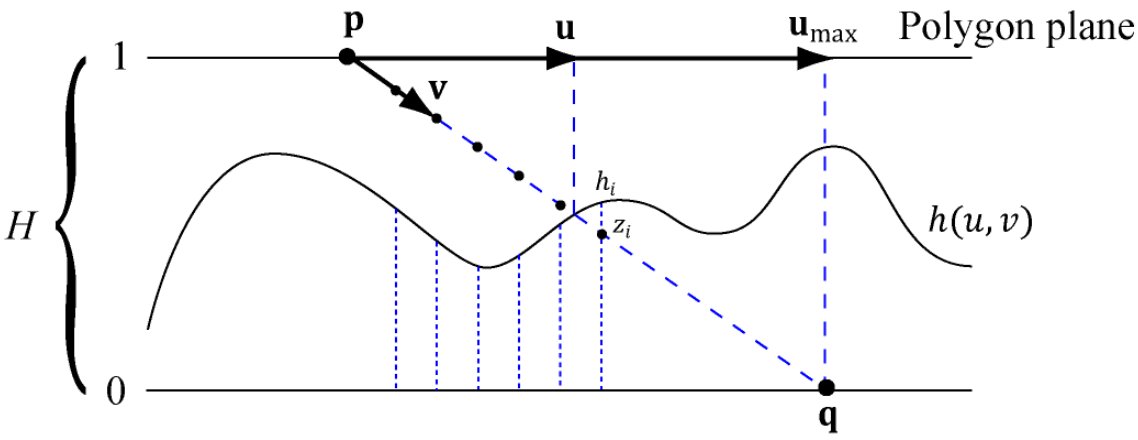

Figure 4: Ray marching with uniform step size. We cross the heightmap the first time $h_i > z_i$.

Note that we ray march in the normalized heightmap space [0, 1]. So if we are taking N samples, we must step a vertical distance $\Delta z = 1/N$ with each step so that the ray travels a vertical distance of 1 by the Nth sample.

Once we find where the ray crosses the heightmap, we approximate the heightmap as a piecewise linear function and do a ray/line intersection test to find the intersection point. Note that this is a 2D problem because we are looking at the cross plane that is orthogonal to the polygon plane that contains the view ray. Figure 5 shows that we can represent the two rays by the equations:

$$\mathbf{r}_1(t) = (0, z_{i-1}) + t(1, z_i - z_{i-1})$$
$$\mathbf{r}_2(t) = (0, h_{i-1}) + t(1, h_i - h_{i-1})$$

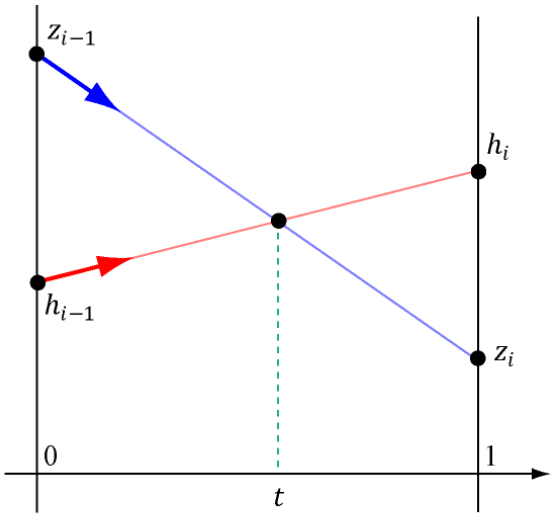

Figure 5: The rays intersect at parameter t.

Observe that in this particular construction, the rays intersect at the same t parameter (in general this is not true). Therefore, we can set the rays equal to each other and solve for t:

$$\begin{aligned}
\mathbf{r}_1(t) &= \mathbf{r}_2(t) \\
(0, z_{i-1}) + t(1, z_i - z_{i-1}) &= (0, h_{i-1}) + t(1, h_i - h_{i-1}) \\
t(1, z_i - z_{i-1}) - t(1, h_i - h_{i-1}) &= (0, h_{i-1} - z_{i-1}) \\
t(0, z_i - z_{i-1} - h_i + h_{i-1}) &= (0, h_{i-1} - z_{i-1})
\end{aligned}$$

In particular, the equation of the second coordinate implies:

$$t = \frac{h_{i-1} - z_{i-1}}{z_i - z_{i-1} - h_i + h_{i-1}}$$

The parameter t gives us the percentage through the interval where the intersection occurs. If the interval direction and length is described by $\Delta\mathbf{u} = \frac{\mathbf{u}_{\max}}{N}$ then the parallax offset vector is given by:

$$\mathbf{u} = (i-1)\Delta\mathbf{u} + t\Delta\mathbf{u}$$

§5 Adaptive Iteration Count

Figure 6 shows that that glancing view rays require more samples than head-on view rays since the distance traveled is longer. We estimate the ray sample by interpolating between minimum and maximum sample count constants based on the angle between the vector from the surface to the eye and surface normal vector:

```
int gMinSampleCount = 8;
int gMaxSampleCount = 64;

int sampleCount = (int)lerp(gMaxSampleCount,
                            gMinSampleCount,
                            dot(toEye, pin.NormalW));
```

For head-on angles, the dot product will be 1, and the number of samples will be `gMinSampleCount`. For parallel angles, the dot product will be 0, and the number of samples will be `gMaxSampleCount`.

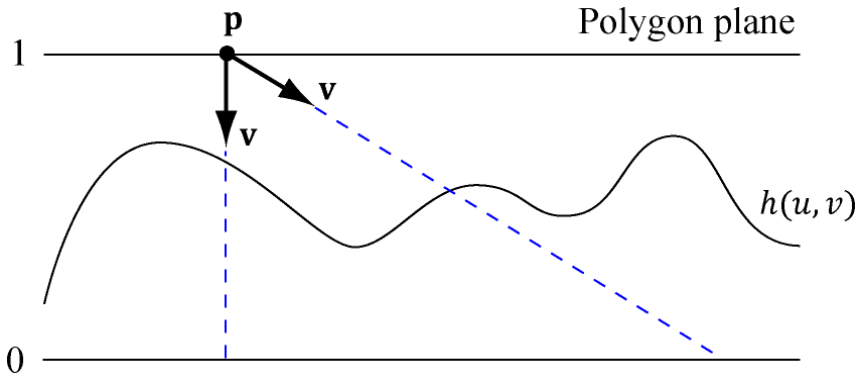

Figure 6: Glancing view rays travel a further distance than head-on view rays. Therefore, we require more samples at glancing angles to get an accurate intersection test.

§6 Code

Our parallax occlusion effect file is given below:

```
#include "LightHelper.fx"

cbuffer cbPerFrame
{
	DirectionalLight gDirLights[3];
	float3 gEyePosW;

	float  gFogStart;
	float  gFogRange;
	float4 gFogColor;
};

cbuffer cbPerObject
{
	float4x4 gWorld;
	float4x4 gWorldInvTranspose;
	float4x4 gWorldViewProj;
	float4x4 gTexTransform;
	Material gMaterial;

	float gHeightScale;
	int gMinSampleCount;
	int gMaxSampleCount;
};

// Nonnumeric values cannot be added to a cbuffer.
Texture2D gDiffuseMap;
Texture2D gNormalMap;
TextureCube gCubeMap;

SamplerState samLinear
{
	Filter = MIN_MAG_MIP_LINEAR;
	AddressU = WRAP;
	AddressV = WRAP;
};

struct VertexIn
{
	float3 PosL     : POSITION;
	float3 NormalL  : NORMAL;
	float2 Tex      : TEXCOORD;
	float3 TangentL : TANGENT;
};

struct VertexOut
{
	float4 PosH     : SV_POSITION;
	float3 PosW     : POSITION;
	float3 NormalW  : NORMAL;
	float3 TangentW : TANGENT;
	float2 Tex      : TEXCOORD;
};

VertexOut VS(VertexIn vin)
```

```
{
	VertexOut vout;

	// Transform to world space space.
	vout.PosW     = mul(float4(vin.PosL, 1.0f), gWorld).xyz;
	vout.NormalW  = mul(vin.NormalL, (float3x3)gWorldInvTranspose);
	vout.TangentW = mul(vin.TangentL, (float3x3)gWorld);

	// Transform to homogeneous clip space.
	vout.PosH = mul(float4(vin.PosL, 1.0f), gWorldViewProj);

	// Output vertex attributes for interpolation across triangle.
	vout.Tex = mul(float4(vin.Tex, 0.0f, 1.0f), gTexTransform).xy;

	return vout;
}

float4 PS(VertexOut pin,
	uniform int gLightCount,
	uniform bool gUseTexure,
	uniform bool gAlphaClip,
	uniform bool gFogEnabled,
	uniform bool gReflectionEnabled) : SV_Target
{
	// Interpolating normal can unnormalize it, so normalize it.
	pin.NormalW = normalize(pin.NormalW);

	// The toEye vector is used in lighting.
	float3 toEye = gEyePosW - pin.PosW;

	// Cache the distance to the eye from this surface point.
	float distToEye = length(toEye);

	// Normalize.
	toEye /= distToEye;

	//
	// Parallax Occlusion calculations to find the texture coords to use.
	//

	float3 viewDirW = -toEye;

	// Build orthonormal basis.
	float3 N = pin.NormalW;
	float3 T = normalize(pin.TangentW - dot(pin.TangentW, N)*N);
	float3 B = cross(N, T);

	float3x3 toTangent = transpose( float3x3(T, B, N) );

	float3 viewDirTS = mul(viewDirW, toTangent);

	float2 maxParallaxOffset = -viewDirTS.xy*gHeightScale/viewDirTS.z;

	// Vary number of samples based on view angle between the eye and
	// the surface normal.  (Head-on angles require less samples than
	// glancing angles.)

	int sampleCount = (int)lerp(gMaxSampleCount, gMinSampleCount,
		dot(toEye, pin.NormalW));

	float zStep = 1.0f / (float)sampleCount;
```

```
float2 texStep = maxParallaxOffset * zStep;

// Precompute texture gradients since we cannot compute texture
// gradients in a loop.  Texture gradients are used to select the right
// mipmap level when sampling textures.  Then we use Texture2D.SampleGrad()
// instead of Texture2D.Sample().
float2 dx = ddx( pin.Tex );
float2 dy = ddy( pin.Tex );

int sampleIndex = 0;
float2 currTexOffset = 0;
float2 prevTexOffset = 0;
float2 finalTexOffset = 0;
float currRayZ = 1.0f - zStep;
float prevRayZ = 1.0f;
float currHeight = 0.0f;
float prevHeight = 0.0f;

// Ray trace the heightfield.
while( sampleIndex < sampleCount + 1 )
{
	currHeight = gNormalMap.SampleGrad(samLinear,
		pin.Tex + currTexOffset, dx, dy).a;

	// Did we cross the height profile?
	if(currHeight > currRayZ)
	{
		// Do ray/line segment intersection and compute final tex offset.

		float t = (prevHeight - prevRayZ) /
			(prevHeight - currHeight + currRayZ - prevRayZ);

		finalTexOffset = prevTexOffset + t * texStep;

		// Exit loop.
		sampleIndex = sampleCount + 1;
	}
	else
	{
		++sampleIndex;

		prevTexOffset = currTexOffset;
		prevRayZ      = currRayZ;
		prevHeight    = currHeight;

		currTexOffset += texStep;

		// Negative because we are going "deeper" into the surface.
		currRayZ -= zStep;
	}
}

// Use these texture coordinates for subsequent texture
// fetches (color map, normal map, etc.).
float2 parallaxTex = pin.Tex + finalTexOffset;

//
// Texturing
//
```

```
// Default to multiplicative identity.
float4 texColor = float4(1, 1, 1, 1);
if(gUseTexure)
   {
          // Sample texture.
          texColor = gDiffuseMap.Sample( samLinear, parallaxTex );

          if(gAlphaClip)
          {
                 // Discard pixel if texture alpha < 0.1.  Note that we do this
                 // test as soon as possible so that we can potentially exit
                 // the shader early, thereby skipping the rest of the shader
                 // code.
                 clip(texColor.a - 0.1f);
          }
   }

   //
   // Normal mapping
   //

   float3 normalMapSample = gNormalMap.Sample(samLinear, parallaxTex).rgb;
   float3 bumpedNormalW = NormalSampleToWorldSpace(normalMapSample,
          pin.NormalW, pin.TangentW);

   //
   // Lighting.
   //

   float4 litColor = texColor;
   if( gLightCount > 0  )
   {
          // Start with a sum of zero.
          float4 ambient = float4(0.0f, 0.0f, 0.0f, 0.0f);
          float4 diffuse = float4(0.0f, 0.0f, 0.0f, 0.0f);
          float4 spec    = float4(0.0f, 0.0f, 0.0f, 0.0f);

          // Sum the light contribution from each light source.
          [unroll]
          for(int i = 0; i < gLightCount; ++i)
          {
                 float4 A, D, S;
                 ComputeDirectionalLight(gMaterial, gDirLights[i],
                        bumpedNormalW, toEye,
                       A, D, S);

                 ambient += A;
                 diffuse += D;
                 spec    += S;
          }

          litColor = texColor*(ambient + diffuse) + spec;

          if( gReflectionEnabled )
          {
                 float3 incident = -toEye;
                 float3 reflectionVector = reflect(incident, bumpedNormalW);
                 float4 reflectionColor  = gCubeMap.Sample(
                        samLinear, reflectionVector);

                 litColor += gMaterial.Reflect*reflectionColor;
```

```
            }
        }

        //
        // Fogging
        //

        if( gFogEnabled )
        {
            float fogLerp = saturate( (distToEye - gFogStart) / gFogRange );

            // Blend the fog color and the lit color.
            litColor = lerp(litColor, gFogColor, fogLerp);
        }

        // Common to take alpha from diffuse material and texture.
        litColor.a = gMaterial.Diffuse.a * texColor.a;

    return litColor;
}
```

§7 Aliasing Issues

Parallax occlusion mapping has aliasing problems as Figure 7 shows. Due to the finite step size, the ray sometimes misses peaks of the heightmap. Increasing the sampling rate (i.e., decreasing the step size) helps, but is prohibitively expensive. We note that tessellation based displacement mapping does not have this problem.

Figure 7: Observe the sampling artifacts.

§8 Extensions and References

[Tatarchuk06] has a detailed description of the algorithm, extends it to support soft shadows and an LOD system, and gives advice for artists authoring heightmaps. [Watt05] devotes a chapter in his book to the algorithm. [Drobot09] describes generating a quadtree structure of the heightmap stored in a texture to improve the performance of the ray/heightmap intersection test.

[Drobot09] Drobot, Michal. "Quadtree Displacement Mapping with Height Blending," GDC Europe 2009. (http://www.drobot.org/pub/M_Drobot_Programming_Quadtree%20Displacement%20Mapping.pdf)

[Tatarchuk06] Tatarchuk, Natalya. "Practical Parallax Occlusion Mapping with Approximate Soft Shadows for Detailed Surface Rendering," *ShaderX5: Advanced Rendering Techniques*. Charles River Media, Inc., 2006.

[Watt05] Watt, Alan, and Fabio Policarpo, *Advanced Game Development with Programmable Graphics Hardware*, AK Peters, Ltd., 2005.