4.2

Multisampling Extension for Gradient Shadow Maps

***Christian Schüler,* Phenomic**

Introduction

ShaderX^4 included the article "Gradient Shadow Maps" [Schueler05], describing a set of techniques to reduce *surface acne*—false shadowed spots on surfaces, mostly at grazing angles in shadow mapping and often incorrectly attributed to a precision problem. The scope of the original article didn't include multisampling, which is often used in shadow mapping to approximate *percentage closer filtering* (PCF), which gives a smoother look to shadow edges. In hindsight this was a significant omission, as the integration of the two techniques is sufficiently not obvious. Therefore, this article is dedicated to bringing multisampling to gradient shadow maps.

Overview of Gradient Shadow Maps

We assume the canonical shadow map algorithm, which renders a depth map from the light's perspective in a first pass and does a comparison against this depth map in a second pass to determine whether a fragment is lit or in shadow [Williams78] and recent overviews in [Moeller02] and [McReynolds05]. In this context, we use *depth* as a measure for the distance from the light source. A pixel shader function to compute the light visibility during the second pass could look like this:

```
float lightVisibility( float3 lightCoord )
{
    return lightCoord.z < tex2D( depthMap, lightCoord.xy ).x;
}
```

Here, `lightCoord` contains the interpolated light space coordinate for the rendered fragment. Without loss of generality, we assume that the light space is affine and omit cluttering up the code with division by *w* (the full shader code on the CD-ROM of course contains this division). The function returns 1 when the fragment is lit and 0 when it is in shadow.

ON THE CD

We observe that this naïve implementation casts false shadows onto half of the rendered surface. See Figure 4.2.1 for an illustration and explanation of why this effect is not a resolution or precision problem.

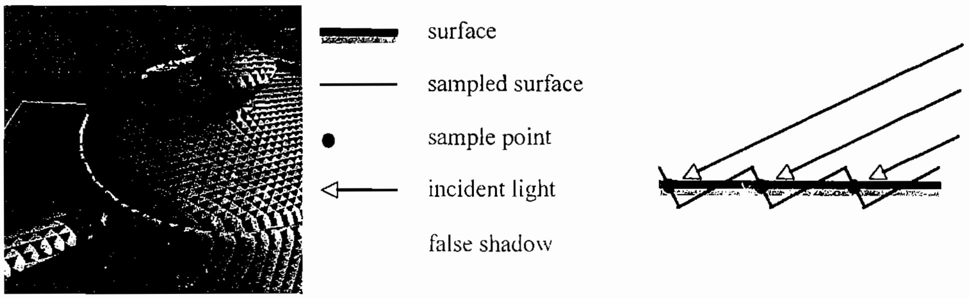

FIGURE 4.2.1 The imperfect reconstruction of the shadow-casting surface gives rise to self-shadowing artifacts.

The well-known remedy for this artifact is to add a small bias value large enough to push the reconstructed surface underneath the reference surface, so we hope no comparisons can fail. However, too much of a bias produces the opposite artifact, called *light bleeding*, at the back face of an occluder. Everyone who has implemented shadow maps will probably remember tweaking the bias value back and forth, alternating between the frying pan and the fire.

Gradient shadow maps address this issue with a combination of three measures that turn out to be mutually beneficial. A central element is the *depth gradient*, which measures the steepness of the surface as seen from the light or mathematically as a vector

$$\nabla z = \left(\frac{\partial z}{\partial x}, \frac{\partial z}{\partial y} \right)$$

with x, y, and z in light view coordinates, z being the depth. This gradient can either be generated from derivative instructions in the first pass and stored with the depth map or estimated from local differences when reading the depth map in the second pass (it may even be calculated from derivative instructions in the second pass if you're willing to solve a set of linear equations). With either method of obtaining a gradient in place, the three improvements are:

Slope-scale depth bias: The bias should be scaled in proportion to the local gradient. This alone can be a potent remedy to some of the nastiest surface acne problems, and it is directly supported for hardware depth maps via the polygon offset feature.

Fuzzy depth comparison: The hard-edged depth comparison (the "less than" operation) should be replaced with a smoother function so that a small numerical error will produce only a small visual error. The range within which the light is attenuated (the fuzzy region) should be proportional to the local gradient.

Linearly filtered depth values: Depth maps are traditionally point sampled. The discontinuous nature of the reconstruction from point sampling is disadvantageous. For instance, if the shadow-casting surface is planar, a bilinear filter over the depth map can reconstruct the original surface perfectly. Further analysis shows that bilinear filtering of the depth map is in no way harmful, even when the underlying surface is not planar. This allows more aggressive reduction of the bias.

The combination of all three measures results in a fuzzy band close underneath the reconstructed surface within which the light visibility gradually falls to zero, allowing for much improved shadow mapping, as shown in Figure 4.2.2. The pseudo code for an improved visibility function might look like this (for the sake of clarity, the code does not consider depth bias):

```
float lightVisibility_GSM( float3 lightCoord )
{
    // get the magnitude of the gradient, by either method
    float gradient = length( getGradientVector( lightCoord ) );
    // get the difference between stored and interpolated depth
    // (depthMap should have LINEAR filtering enabled)
    float diff = tex2D( depthMap, lightCoord.xy ).x -
    lightCoord.z;
    // replace the less-than operator with a smooth function
    // for diff >= 0 the result is 1
    return saturate( diff / gradient + 1 );
}
```

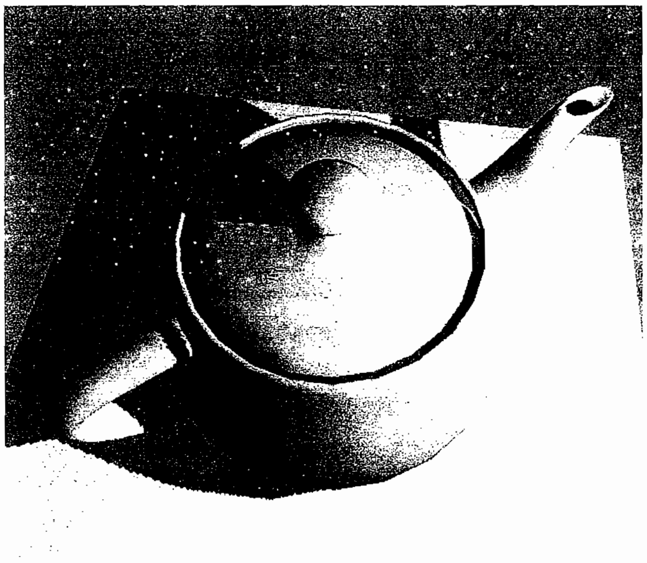

FIGURE 4.2.2 Artifact-free shadow mapping with a just a tiny amount of depth bias needed.

Multisampling and Percentage Closer Filtering

If we consider a given area on the depth map, the term *percentage closer filtering* refers to estimating the fraction of that area that is closer to the light than a given reference depth [Reeves87]. This can be implemented either as an integral over an implied continuous function (which is what bilinear PCF in hardware effectively reduces to) or as a discrete sum over a set of individual samples. In the latter case it is simply the number of depth comparisons that succeed. A visibility function that implements PCF via direct summation may look like this:

```
float lightVisibility_PCF( float3 lightCoord )
{
    float result = 0;
    for( int i = 0; i < n; ++i )
    {
        float3 offCoord = lightCoord + offset[i];
        result += lightCoord.z < tex2D( depthMap, lightCoord.xy ).x;
    }
    return result / n;
}
```

Here, the number of samples taken is n, the light space coordinate is displaced by some offset[] before texture lookup takes place, and the results are averaged. PCF improves shadow mapping by making shadow edges appear less jagged and aliased.

As can be seen from the code above, it would make perfect sense to substitute the less-than operator for any other visibility function, with possible outcomes between 0 and 1. In this case the PCF algorithm is no longer counting a closer-than relation. Instead it accumulates an average visibility, so it really should be called along the lines of *average visibility filtering*. For this article we will stick with the term *PCF*.

Merging the Algorithms

As outlined above, PCF can be seen as an average of light visibility, so it is natural to construct an algorithm that averages the results of our GSM visibility function:

```
float lightVisibility_PCF_with_GSM( float3 lightCoord )
{
    float result = 0;
    for( int i = 0; i < n; ++i )
        result += lightVisibility_GSM( lightCoord + offset[i] );
    return result / n;
}
```

Much to our dismay, the result of a simple code substitution is not what we were expecting. Instead of smoothed shadow edges, it looks like surfaces are now covered with a dark layer that approximately halves the light intensity (see Figure 4.2.3).

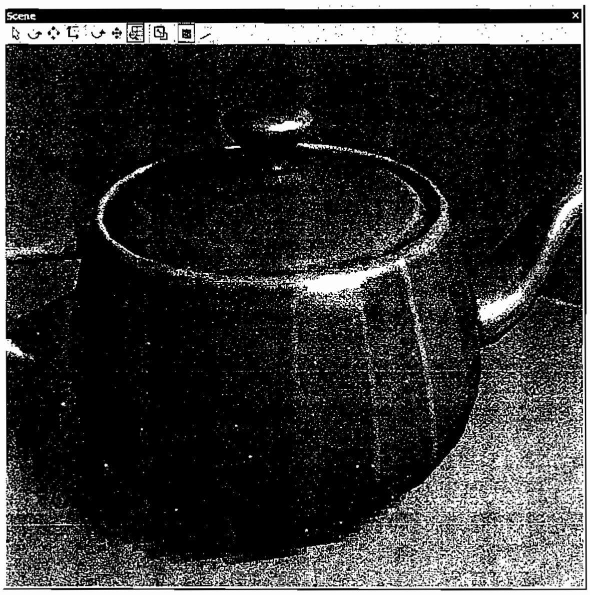

FIGURE 4.2.3 Reduced light intensity from PCF disk intersection with the shadow volume.

Further analysis shows what has gone wrong. As in traditional PCF, the offset added to the interpolated light space coordinate only affects texture coordinates *x* and *y*, not the depth. In this case a constant depth is tested against samples from different locations. Without a sufficient depth bias, the PCF area will eventually intersect the shadow volume (see Figure 4.2.4a). Since the depth biases used with GSM are very small, the PCF almost always intersects the shadow volume, and the average light intensity is reduced. This is the cause of the visual disruption seen in Figure 4.2.3.

A correction is needed. When offsetting the *x* and *y* coordinates, we must also offset the depth coordinate *z* to follow the local depth gradient and prevent intersection with the shadow boundary, as shown in Figure 4.2.4b. This is effectively following a plane that extends from the local surface. A modified PCF framework therefore needs to make the offset known to the visibility function, so that the visibility function can make the necessary adjustments. The offset will become the second parameter to the visibility function:

```
float lightVisibility_PCF_modified( float3 lightCoord )
{
    float result = 0;
    for( int i = 0; i < n; ++i )
        // forward the offset into the visibility function
        result += lightVisibility_GSM_modified( lightCoord,
        offset[i] );
    return result / n;
}
```

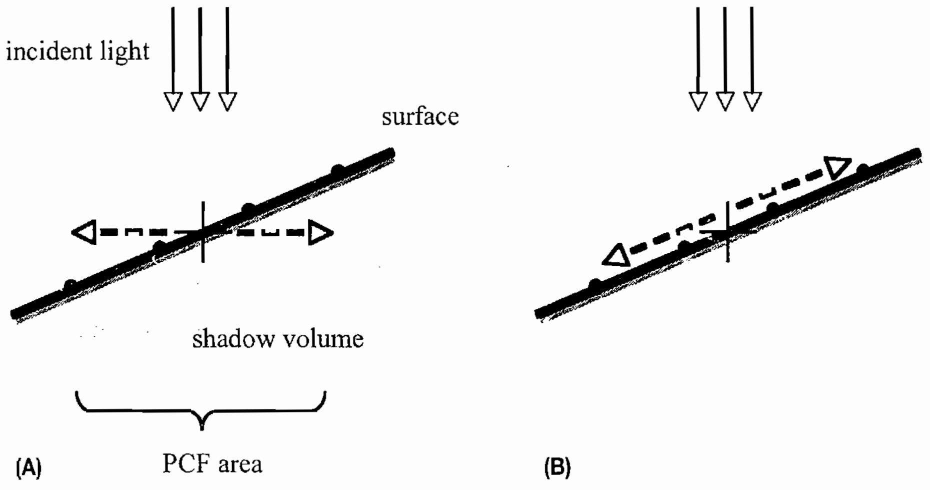

FIGURE 4.2.4 Uncorrected (**A**) and slope-corrected (**B**) offset lookup for PCF. The crosshair indicates the position of the fragment being rendered.

A modified visibility function will use the offset to adjust the depth *z*. The gradient vector points in the direction of increasing depth, while the magnitude of the gradient vector is the slope. Therefore, a simple dot product between the offset vector and the gradient vector will give the desired *z* offset:

```
float lightVisibility_GSM_modified( float3 lightCoord, float2
offset )
{
    // get the gradient, by either method
    float2 gradientVector = getGradientVector( lightCoord );
    float gradient = length( gradientVector );

    // calculate an offset coordinate
    // the z coordinate is moved along with the local gradient
    // (this is equivalent to having a local plane equation)
    float3 offCoord = float3(
        lightCoord.xy + offset,
        lightCoord.z + dot( offset, gradientVector ) );

    // the rest is straightforward
    float diff = offCoord.z - tex2D( depthMap, offCoord.xy ).x;
    return saturate( diff / gradient + 1 );
}
```

Finally, we need an analysis of what happens when the modified PCF algorithm encounters a concave surface. As shown in Figure 4.2.5a, the entire PCF area is underneath the concave surface, and the outer regions of the PCF area touch the shadow volume, causing self-shadowing artifacts. Whether the PCF area reaches into the shadow volume or not depends on the curvature of the surface and the bias and range of the shadow boundary, but since the surface is no longer planar, it is clear that a simple linear extension of the PCF area is insufficient. There is no upper limit on the curvature of a concave surface, so any amount of biasing might eventually fail. In practice, this effect results in dark contours at concave polygon intersections, which can be seen as rings around the neck of the knob in Figure 4.2.5b. The obvious improvement is to consider higher-order derivatives to better approximate the local curvature. Another possible improvement is to use the average gradient of the entire PCF area, rather than the gradient at the queried point. As shown in Figure 4.2.5, using the average gradient results in less severe intersections with the shadow volume than if the PCF area followed the gradient of the point where the crosshair is.

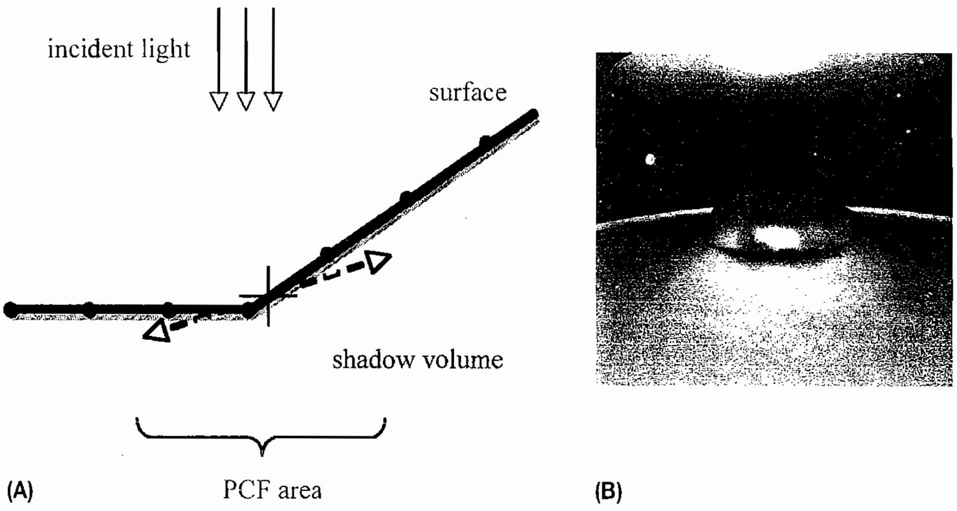

FIGURE 4.2.5 Schematic (A) and example (B) of PCF look-up at a concave intersection. The crosshair indicates the position of the fragment being rendered.

Optimizing Texture Look-ups

All code samples so far have been presented with clarity in mind, not performance. There is much redundancy and opportunity for optimization. If we do not store the gradient vector with the depth map, but only the scalar depth, then we need to perform at least three texture look-ups to compute the gradient vector. Add one texture look-up for the depth sample itself, and we have four look-ups per PCF iteration. Texture look-ups are an expensive resource and should be used more economically.

Since we would like an average gradient over of the entire PCF area, it is possible to use the depth samples to calculate it. What we're trying to do is to estimate a plane equation from a cloud of point samples. The canonical tool for this task is a *linear regression* to find a least squares approximation. We're not going to do a full regression in our shader code, but the theoretical foundation serves us to understand the shortcuts we're going to take. Consider the following closed-form solution for a two-dimensional regression centered around zero, where Δx_i and Δy_i are components of offset vectors $\Delta \mathbf{p}_i$ while Δz_i is depth differences along corresponding offset vectors:

$$\begin{bmatrix} \partial z/\partial x \\ \partial z/\partial y \end{bmatrix} = \begin{bmatrix} \sum \Delta x_i^2 & \sum \Delta x_i \Delta y_i \\ \sum \Delta x_i \Delta y_i & \sum \Delta y_i^2 \end{bmatrix}^{-1} \begin{bmatrix} \sum \Delta z_i \Delta x_i \\ \sum \Delta z_i \Delta y_i \end{bmatrix}$$

$$\downarrow$$

$$\nabla z = \left(\sum \Delta \mathbf{p}_i \Delta \mathbf{p}_i^T\right)^{-1} \sum \Delta z_i \Delta \mathbf{p}_i .$$

The column vector on the left is the estimate for the gradient vector. The matrix in the middle is the *covariance matrix*. The column vector on the right contains the offset vectors weighted by their corresponding depth difference. Observe that the covariance matrix can be diagonalized if the off-diagonal terms $\sum \Delta x_i \Delta y_i$ vanish. This is the case, for instance, if all offset vectors are either horizontal or vertical or if the point cloud is circular symmetric. The inverse of a diagonal matrix is then a diagonal matrix with inverted elements, and the equation becomes

$$\begin{bmatrix} \partial z/\partial x \\ \partial z/\partial y \end{bmatrix} = \begin{bmatrix} \sum \Delta z_i \Delta x_i / \sum \Delta x_i^2 \\ \sum \Delta z_i \Delta y_i / \sum \Delta y_i^2 \end{bmatrix}.$$

Figure 4.2.6a shows an example of eight PCF samples arranged in a circular pattern where we can use the simplified regression formula. The arrows in the diagram indicate suitable paths for differentiation; these are our Δx_i and Δy_i (they need not be rooted at the center). To efficiently compute an average gradient vector according to this scheme, the loop of the PCF function is unrolled and differences are assigned to components of the gradient vector:

```
float lightVisibility_unrolled_circular8( float3 lightCoord )
{       const float displace = .5 / depthMapResolution;
    const float displaceLong = 1.41421 * displace;
    float depths[8] = {
        tex2D( depthMap, lightCoord.xy + float2( -displaceLong,
        0 ) ).x,
        tex2D( depthMap, lightCoord.xy + float2( displaceLong,
        0 ) ).x,
        tex2D( depthMap, lightCoord.xy + float2( 0,
        -displaceLong ) ).x,
        tex2D( depthMap, lightCoord.xy + float2( 0,
        displaceLong ) ).x,
        tex2D( depthMap, lightCoord.xy + float2( -displace,
        -displace ) ).x,
        tex2D( depthMap, lightCoord.xy + float2( displace,
        -displace ) ).x,
        tex2D( depthMap, lightCoord.xy + float2( -displace,
        displace ) ).x,
            tex2D( depthMap, lightCoord.xy + float2( displace,
            displace ) ).x
        };
        const float inverseCovariance = 1. / ( 1 + 1 + 2 );
        float2 gradientVector = float2(
              depths[1] * 1.41421 + depths[5] + depths[7]
            - depths[0] * 1.41421 - depths[4] - depths[6],
              depths[3] * 1.41421 + depths[6] + depths[7]
            - depths[2] * 1.41421 - depths[4] - depths[5] ) *
            inverseCovariance;

        // continue with summation over PCF samples using the
        // average gradient
    }
```

Basically, each sample contributes to each component of the gradient vector based on the sign of the corresponding component in the offset, while the squares of the path lengths are collated into a single renormalizing factor. Observe that no additional texture lookups are needed besides the ones already used for the PCF samples, unlike the previous case if we had stored the gradients into the depth map.

If the PCF pattern is irregular, for instance, if the sample locations constitute a Poisson disk, it would be possible to precompute the inverse of a covariance matrix. The most straightforward solution to calculate a regression is then to sum the products of all or some of the depths with their corresponding offset vectors and multiply the result with the precomputed covariance matrix. If the PCF pattern is sufficiently circular, the covariance matrix may even turn out to be close to diagonal, reducing it to a scale factor. An alternative method is sort of a hack: to use the simplified regression formula in this case while staying approximately accurate, select some difference paths in the point cloud that already are almost horizontal or vertical and then weight them so that their vector sum becomes aligned. For instance, to compute the *x*-component gradient vector from a large irregular PCF pattern, select a number of difference paths already approximately horizontal and weight them such that their *y*-component vanishes, as is illustrated in Figure 4.2.6b.

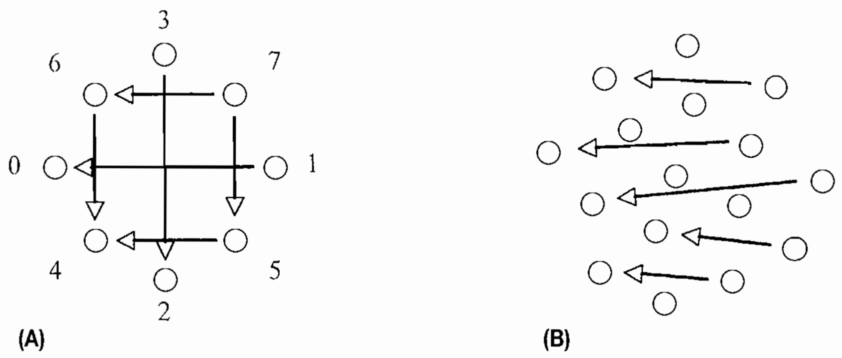

FIGURE 4.2.6 Difference paths in regular (A) and irregular (B) PCF patterns.

Optimizing with SIMD Vectorization

The shading languages of today's graphics processors are structured around four-way *single instruction multiple data* (SIMD) instructions. With proper SIMD vectorization, batches of similar operations can be calculated in parallel, and summation and averaging can be performed by dot products. We begin by loading the PCF samples into a two-array of four-vectors, which does not look very different from the previous version:

```
float lightVisibility_unrolled_circular8_SIMD( float3 lightCoord )
{
    const float displace = .5 / depthMapResolution;
    const float displaceLong = 1.41421 * displace;
    float4 depths[2] = { float4(
        tex2D( depthMap, lightCoord.xy + float2( -displaceLong,
        0 ) ).x,

        tex2D( depthMap, lightCoord.xy + float2( displaceLong,
        0 ) ).x,
        tex2D( depthMap, lightCoord.xy + float2( 0,
        -displaceLong ) ).x,
        tex2D( depthMap, lightCoord.xy + float2( 0,
        displaceLong ) ).x
    ), float4(
        tex2D( depthMap, lightCoord.xy + float2(
        -displace, -displace ) ).x,
        tex2D( depthMap, lightCoord.xy + float2( displace,
        -displace ) ).x,
        tex2D( depthMap, lightCoord.xy + float2( -displace,
        displace ) ).x,
        tex2D( depthMap, lightCoord.xy + float2( displace,
        displace ) ).x
    ) };
...
```

Observe that the sample that used to be depths[0] is now depths[0].x, the sample formerly in depths[1] is now depths[0].y, and so on. The first optimization is to compute both components of the gradient vector in parallel:

```
    const float inverseCovariance = 1. / ( 1 + 1 + 2 );
    float2 gradientVector = float2(
          depths[0].yw * 1.41421 + depths[1].yz + depths[1].ww
        - depths[0].xz * 1.41421 - depths[1].xx - depths[1].zy )
              * inverseCovariance;
...
```

The second optimization is to parallelize the computation of the depth differences. Recall that in this step we need to move the light space coordinate along the local gradient to the offset position and then subtract it from the depth found there. The dot products between the offset and gradient vectors can be collated into a series of 2 × 4 matrix multiplies. This step produces eight depth differences, stored in two four-vectors, `diffs[0]` and `diffs[1]`:

```
        // these matrices mirror the offsets in convenient ordering

        const float2x4 offsData[2] = {

            float2x4( float4( -1, 1, 0, 0 ), float4( 0, 0, -1, 1 ) )
            * 1.41421,

            float2x4( float4( -1, 1, -1, 1 ), float4( -1, -1, 1, 1 )
            ) };


        // aggregate dot( offset, gradientVector ) done by mul(...)
        float4 diffs[2] = {
            depths[0] - lightCoord.z - mul( gradientVector,
            offsData[0] ),
            depths[1] - lightCoord.z - mul( gradientVector,
            offsData[1] ) };
    ...
```

In the last step the calculation of the visibility function is done in parallel for batches of four PCF samples. To accumulate the result and return an average visibility, we can use dot products by observing that a dot product of a four-vector with the vector (1,1,1,1) is in effect the sum of its components, while a dot product with the vector (0.25,0.25,0.25,0.25) is the average of its components. Therefore, we use a series of four-vector and two-vector dot products to compute the average of the eight visibility functions:

```
        return dot( .5, float2(
            dot( .25, saturate( diffs[0] / gradient + 1 ) ),
            dot( .25, saturate( diffs[1] / gradient + 1 ) ) ) );
    }
```

ON THE CD

The shader code on the CD-ROM contains an implementation of this function augmented with adjustable bias and adjustable radius of the PCF pattern.

Conclusion

This article showed how to add multisampling to gradient shadow maps to achieve a smoother look for shadow edges. We ended up with a substantial reduction in depth aliasing and a good-looking shadow penumbra.

References

[McReynolds05] McReynolds, Tom and Blythe, David. *Advanced Graphics Programming Using OpenGL.* Morgan Kaufmann Publishing, San Francisco, 2005.

[Moeller02] Möller, Tomas and Haines, Eric. *Real-time Rendering.* A K Peters Ltd, Natick, MA, 2002.

[Reeves87] Reeves, W. T., D. H. Saselin, and R. L. Cook. "Rendering Antialiased Shadows with Depth Maps." *ACM SIGGRAPH Computer Graphics, 21,* (1987): pp. 283–291.

[Schueler05] Schüler, Christian. "Eliminating Surface Acne with Gradient Shadow Mapping." *ShaderX^4*, edited by W. Engel. Charles River Media, Hingham, MA, 2005.

[Williams78] Williams, Lance. "Casting Curved Shadows on Curved Surfaces." *Proceedings of the 5th Annual Conference on Computer Graphics and Interactive Techniques,* 1978: pp. 270–274.