Fractional-Disk Soft Shadows

Michal Valient
Dep. of Computer Graphics and Image Processing
Comenius University in Bratislava

Willem H. de Boer
Graphics Research Group
Playlogic Game Factory BV

Introduction

This chapter will describe a simple, practical, and fairly effective way to approximate soft shadows. The actual implementation is a slight modification of percentage closer filtering (PCF) [Reeves87, Bunnell04]. PCF was originally designed to address the aliasing problem inherent in shadow mapping. It also has the welcome side effect of generating soft shadow edges, which resemble simple constant-width penumbrae. In this chapter we will show how a slightly modified PCF algorithm can produce more realistic penumbrae. We will use stochastic sampling varied per pixel to obtain reasonable results while using fewer shadow map samples.

Previous work

Our algorithm extends the well-known shadow mapping algorithm published by Lance Williams back in 1978 [Williams78]. This two-pass algorithm is currently popular because of its simplicity and execution speed on modern graphics hardware. Its main disadvantage is that it suffers from aliasing problems because of the finite resolution and precision of the shadow map. Reeves et al. proposed a method for anti-aliasing shadow maps called percentage closer filtering [Reeves87]. The idea is to first perform a number of depth tests in a certain region of lightspace using unfiltered samples from the shadow map, and then use this information to calculate the percentage of shadow this region receives. Performing this type of anti-aliasing can be done by the GPU [Bunnell04].

We refer the interested reader to the summary report by Hasenfratz et al. [Hasenfratz03] for more background information about various real-time soft shadow algorithms.

Soft shadows

Our algorithm considers four types of regions: fully lit, umbra, and two types of penumbra, which are divided by the original hard shadow boundary (see Figure 1a). In one penumbra region (P2) the shadow increases and borders the fully shadowed umbra region; this is the inner penumbra region. In the other region (P1) the shadow intensity decreases and ends in full light; this is the outer penumbra region. The point Pe divides P1 from P2, and is where the hard shadow falls for a point light at the center of the area light (i.e. the hard shadow boundary).

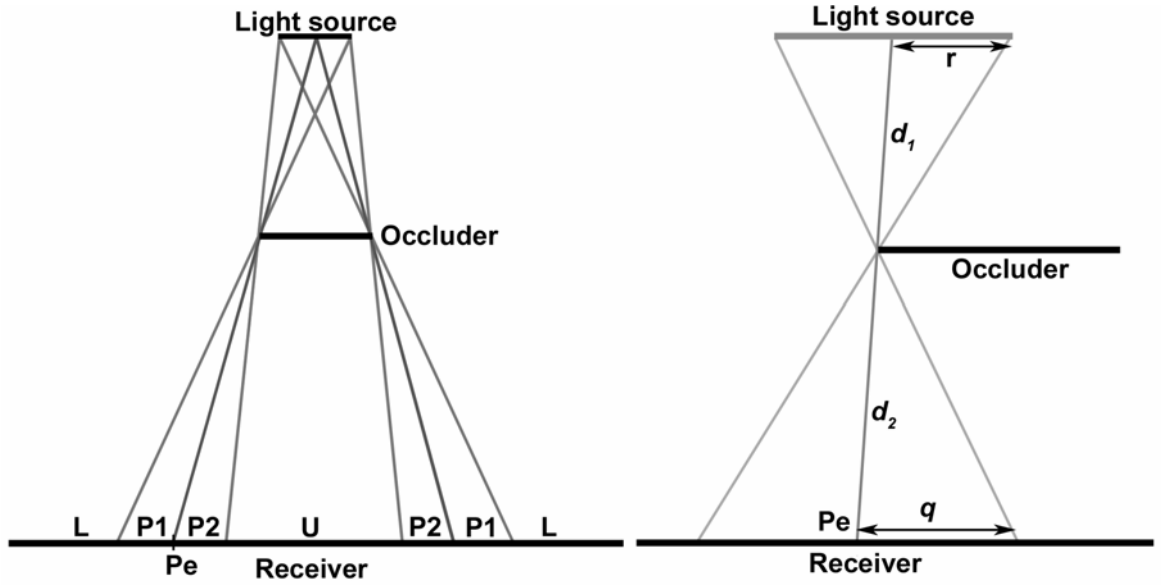

Figure 1a: Four types of shadow regions – umbra (U), inner penumbra (P2), outer penumbra (P1), full light (L).
Figure 1b: Penumbra region radius computation components.

Now consider Figure 1b. The receiver is considered to be of constant z in lightspace (i.e. it is perpendicular to the light's shadow map direction). The circular lightsource casts inner and outer penumbra regions onto the receiver. We shall consider an arbitrary point P on the receiver. The key observation is that we can convert the effect of an area light on a lit point as being approximately equivalent to the effect of a point light on a lit disk. The amount of light that P receives from the lightsource is proportional to the fraction of the area of a disk D[1] with radius *q* and center P is being lit by a point lightsource located at the center of the area lightsource. This means that to calculate the amount of light arriving at any point P on the receiver, we simply construct a disk of radius *q* with center at P and facing the light, and divide the area lit by the total area of the disk. This gives us a scalar value in the range [0.0, 1.0], where 0.0 means completely shadowed and 1.0 means completely lit. Note how with this 2D construction the point Pe in Figure 1b will be classified as having a shadow value of 0.5.[2]

The radius *q* of the disk for an arbitrary point on the receiver is computed from the radius of circular light source *r*, the difference in light space depth between the lightsource and the occluder d_1 and the difference in depth between the receiving point and the occluder d_2, using Notice the geometrical relationship between disk radius *q*, the difference in depth between receiver and occluder d_2, the difference in depth between occluder and lightsource d_1, and the radius of the lightsource *r*, as shown in Equation 1:

$$q = \frac{d_2}{d_1} r \tag{1}$$

We need to stress the fact that *q* is the same for *every* point P on the constant z receiver in Figure 1b; its value can be calculated by using Eq. 1 as if we were calculating *q* for Pe. Intuitively, this means that to calculate the disk radius for any point P (even on receivers of

[1] Note that if L were star-shaped, for example, this disk would instead be a star-shaped region.
[2] In 3D, points like Pe do not always have a shadow value of 0.5, e.g. at a convex silhouette vertex the value is higher than 0.5, at a concavity it is less than 0.5.

non-constant z), we must first find the corresponding point Pe that lies in the *same imaginary plane of constant z* as P, and calculate *q* using that. This turns out to be difficult to do in practice, therefore our actual implementation allows only an approximation of the actual disk radius to be computed by far easier means. However, we will describe a slightly more accurate method in the subsection “Implementation without preprocessing”.

Stochastic sampled percentage closer filtering

In the previous section we showed how to compute the amount of shadow a point P receives by considering the ratio between the lit part of a disk centered at P and the total area of this disk. In the discrete case we can implement this by picking a finite number, N, of samples in the disk and finding how many of these samples are lit. This is essentially equivalent to performing percentage closer filtering in terms of the algorithm’s behavior. The main change we make is that the radius *q* varies as *d1* and *d2* change, whereas with the standard PCF approach this radius is kept more or less fixed[3]. Increasing the radius of the penumbra region means the disk covers more shadow map texels and so we need to enlarge the filtering kernel for the PCF appropriately. However, the resulting increase in the number of samples means a slower processing speed. To compensate for this we will process only a fraction of the texels in the region.

For a given receiver point we randomly choose some texels within the disk region, and perform PCF using these. The main advantage of this method is that for the average case we get acceptable results with less computational cost. The disadvantage is that the shadows are less precise. We use less texels, so we lower the Nyquist limit of the sampled shadow data, which could result in aliasing [Cook86]. This problem is not that serious overall, since soft shadows are normally low-frequency by nature. The main objectionable drawback is that any aliasing has been replaced by noise due to stochastic sampling. This manifests itself as noisy penumbrae. However, this looks better than the banding artifacts that would otherwise occur.

One of our goals is to do as little computation as possible, so we choose the number of samples based on the disk radius *q*. Equation 2 shows the function that returns the number of samples depending on the disk area (with radius *q*).

$$f(q) = \left\lceil \frac{\pi}{a} q^2 \right\rceil \tag{2}$$

The constant *a* specifies the ratio between the real sample count and the desired sample count. Of course, other equations are possible. Figure 2 shows PCF sampling kernels based on Equation 2.

[3] In the original paper, this region is not really kept fixed, but changes with the size of the particular cameraspace region.

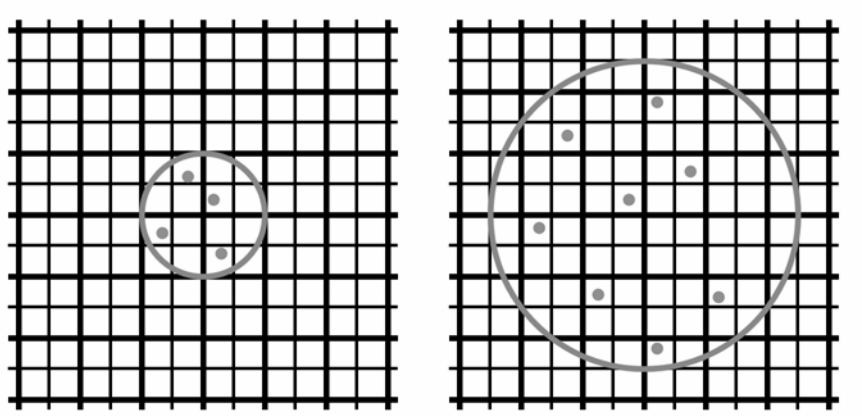

Figure 2: PCF kernel based on Equation 2 for q=2, a=2 (left) and q=5, a=2 (right).

This sampling method works only for points on receivers that are of constant z in light space (essentially, perfectly facing the light). To see an example of when the technique fails, consider the left side of Figure 3. The point P has now been chosen to lie well outside the shadowed regions. However, a problem occurs because the disk centered about P penetrates the receiver plane and is therefore partially shadowed by the receiver itself. Brabec et al. [Brabec02a] solve this problem by assigning unique IDs to objects, which allows them to discard occlusion information whenever the ID of a point on the receiver matches the ID of the point occluding it. Unfortunately, this solution has the effect that objects cannot cast soft shadows onto themselves. If instead we store the back-faces rather than the front-faces in the shadow map the problem goes away. This method is also known as second-depth shadow mapping and is described in [Wang97] and [Zioma03], where it is used to solve a similar problem.

Unfortunately, the same artifact is now relegated to disks that cross back-facing surfaces, an example of which is shown in Figure 4. Consider the right side of Figure 3, which is a cross-section of a simple scene consisting of two solid objects. The thick gray lines indicate those points on the objects that are stored in the actual shadow map (they are the nearest back-faces as seen from the lightsource). Point P_2 (which happens to lie in a penumbra region in this particular case) has its disk illustrated by the black horizontal line. Notice how this disk penetrates the receiver object; all points in the disk that are to the right of P_2 are *incorrectly* classified as being occluded by the receiver itself. We solve this by considering these points in the disk as not occluded, as follows. We will take C as an example point. The same test needs to be performed for every point in the disk. We construct a disk around C, by using Eq. 1 (where d_1 is the distance between Cs and the lightsource and $d2$ is I_2). If P_2 is *not* contained in this new disk (i.e. is outside the radius r_2) we know that C was *incorrectly* classified as occluded, and we therefore count it as not occluded instead. Note how with this construction, point A, which is *correctly* classified as occluded, will remain occluded by this additional test, because its disk contains P_2, i.e. P_2 is inside A's disk of radius r_1

A similar but less disturbing artifact is illustrated in Figure 3c. Point B (which is inside the object) in the disk centered at P is now incorrectly classified as not occluded. This results in point P being assigned a higher PCF (i.e. less in shadow) value than it should actually get.

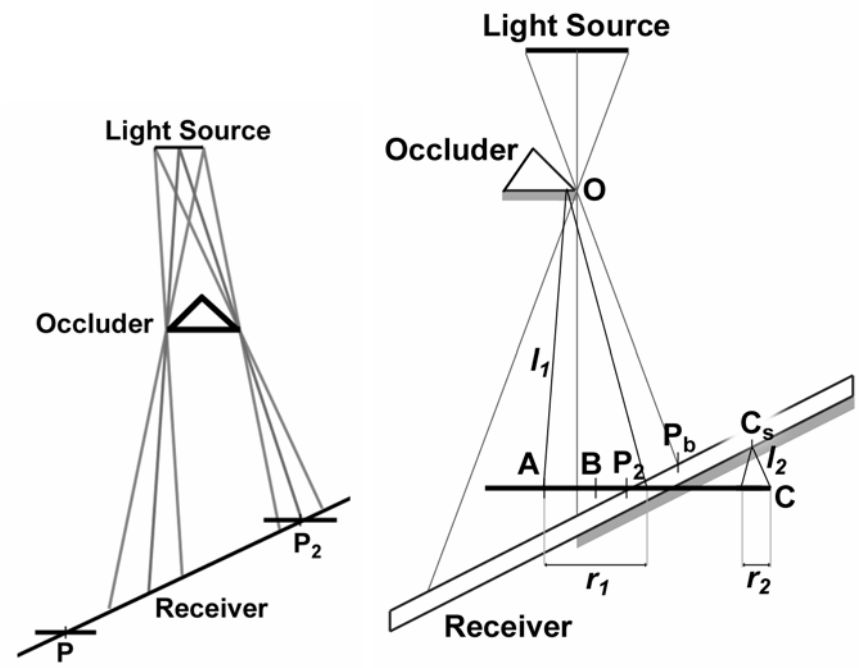

Figure 3a: A side view of a receiving plane of non-constant z in lightspace (left). Notice how the disk (which *is* of constant z) with center P penetrates the plane.
Figure 3b: The penetrating disk problem (right).

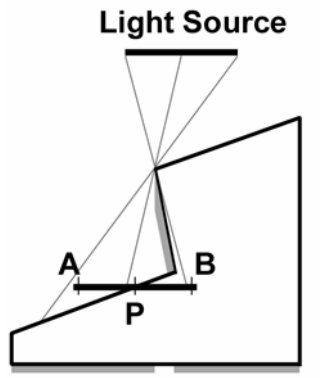

Figure 3c: Point P is assigned a higher PCF value than it should really get (i.e. is brighter than it should be), as point B on the disk has been incorrectly classified as not occluded.

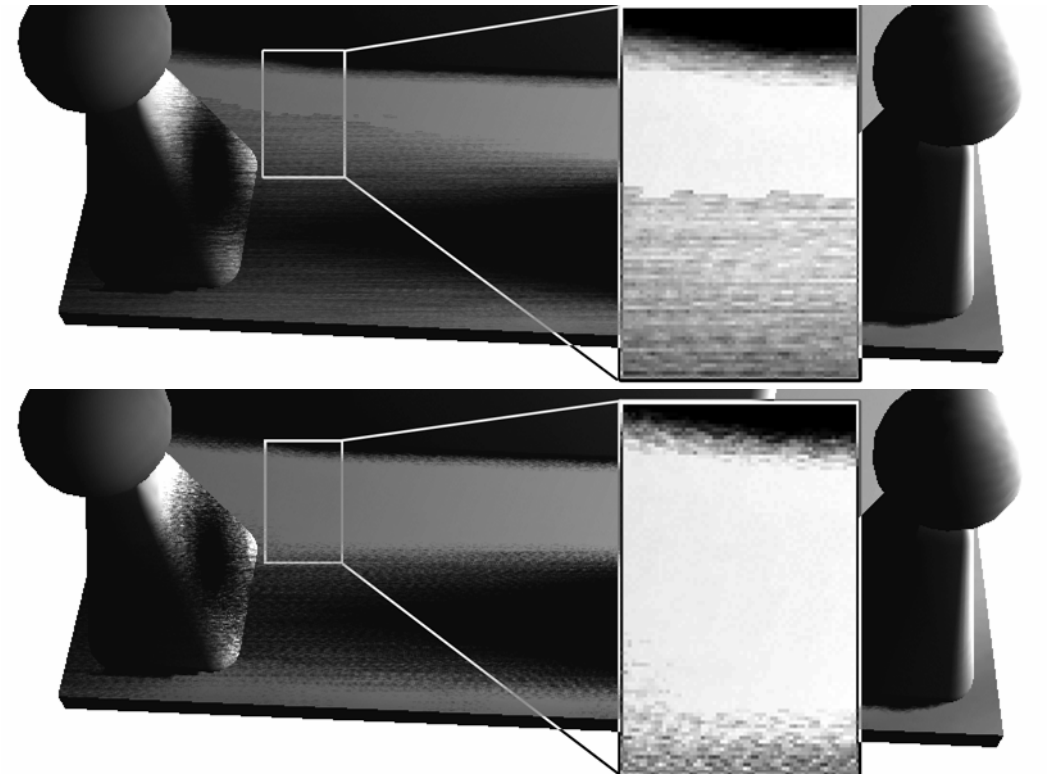

Figure 4: The upper figure shows self-shadowing that occurs in the outer penumbra region, because the slab is thin and its backface shadows the sample disks. The bottom shows the (more) correct outer penumbra regions once our two-step method is applied. The brightness and contrast of the zoomed-in areas was modified so the problem is more visible.

Implementation

In this section we will describe the ps_3_0 version of our implementation, which features loops and flow control, which are the main components needed for the full version of our algorithm. We will outline the implementation steps in detail, but avoid the use of shader code listings in favor of higher level pseudo-code. On the CD you can find the fully documented source code using HLSL for ps_3_0, ps_2_a (ps_2_b), and a limited version for ps_2_0 hardware.

There are four major parts of the algorithm: noise map generation, shadow map generation, shadow map preprocessing and final rendering.

Noise map generation

To obtain random coordinates during rendering we use a special noise map, which can be of arbitrary dimensions. This noise map needs to be generated only once. Each texel of this map contains random values in the range [0.0, 1.0] in each color channel. The red and green channels are used to modify the original shadow map coordinates, and they have to meet the following constraint that makes them lie in the unit disk:

$$(2texel.r - 0.5)^2 + (2texel.g - 0.5)^2 \leq 1 \qquad (3)$$

During rendering these coordinates will be scaled by the disk radius so they fit in the actual disk.

Each row of values in the red and green channels of the map is generated using a Poisson disk distribution (see Cook [Cook86] and the chapter “Poisson Shadow Blur” on page ?? for more on this subject). During the final rendering step, explained below, we sequentially sample the rows of the map to obtain disk sample coordinates.

The blue and alpha channels contain random values without the constraint in Equation 3, and we use these only once per pixel to compute the initial random coordinates used later for the sequential reading of the red and green channels, as will be explained in the final rendering step.

Shadow map generation

This step is almost identical to the standard shadow mapping algorithm, but with two small changes. First, we render only polygons facing away from the light. Second, the shadow map does not contain the lightspace *z* coordinates. Instead we compute for each vertex the distance to the light in world space, linearly interpolate this distance across the triangle, and store this in the shadow map. Values in the light’s attenuation range are mapped linearly into the range [0.0, 1.0] using Equation 4 (where d_{light} is distance from light in world space, $l_{AttenuationStart}$ is the lowest allowed distance from light, and $l_{AttenuationEnd}$ is the maximum allowed distance from the light). This is described in more detail by Valient [Valient03]. Brabec describes an even more clever computation of linear distance [Brabec02b].

$$ShadowMap.r = \frac{d_{light} - l_{AttenuationStart}}{l_{AttenuationEnd} - l_{AttenuationStart}} \tag{3}$$

Shadow map preprocessing

The distance stored in the shadow map does not provide us with enough information. The problem is that points in outer penumbra region P1 have no information about the nearest occluder in the shadow map – they are considered lit as far as the shadow map algorithm is concerned. Therefore we perform a preprocessing step where we add this missing information to the shadow map. For each texel T of the shadow map that can -- during final rendering -- be mapped to a point in region P1 we store the depth of the occluder nearest (in the lightspace x and y dimensions) to T in the shadow map. During final rendering we use this value to estimate the approximate difference in depth between T and its occluder. This difference is then used to compute the disk radius for T.

Points that lie either in the umbra or in the completely lit region should be processed as quickly as possible; the PCF operation for these points is redundant, and should therefore be avoided. To accomplish this we store for each texel in the shadow map the distance to the nearest hard shadow boundary (the point Pe in Figure 1b). This information will be used to identify such points as being too far away, which allows us to skip the costly PCF operation for these points.

Preprocessing is done in multiple passes and the final shadow map will contain the original depth values in the red channel, the depth of the nearest occluder in the green channel, and the distance to the nearest shadow boundary in the blue channel.

To implement this, we start by applying an edge detection filter to the shadow map. The resulting edges specify the hard shadow boundaries (point Pe in Figure 1b). If a texel is classified as belonging to such an edge we store its depth value in the green channel, otherwise we store 1.0 (the maximum depth) in the green channel. Because the edge detection filter can (and in practice will) also mark some texels near the real edge, we do not directly copy the depth value from the red channel to the green channel. Rather, we choose the minimum of the depth values from neighboring texels. For edge texels we also store a value of one distance unit into the blue channel (this is 1/256 for shadow map with resolution 256x256). In subsequent passes we will use this value to accumulate the distance from the edge. The following pseudo-code illustrates the first pass.

```
For each shadow map texel t
{
    bool bIsEdge = DetectEdge(t);
    if (bIsEdge)
    {
        float fMinDepth = GetMinimumAreaDepth(t);
        t = float4(t.x, fMinDepth, fDistanceUnit, 0);
    }
    else
        t = float4(t.x, 1.0f, 0, 0);
}
```

For each texel in subsequent passes we check for a value other than 1.0 in its green channel and that of its neighbors. We use the minimum of these depths and store it in the green channel of the texel.

We also compute the maximum of the blue channel values. If the maximum is greater than zero we add one distance unit to this value and store it in the blue channel.

By repeating this pass several times we distribute the occluder depth into the green channel and accumulate an approximate (inverse) distance to the nearest edge in the blue channel. The exact number of passes depends on the largest allowed penumbra radius.

```
For each shadow map texel t
{
    //Get average depth (ignore 1.0f)
    float fDepth = GetMinimumAreaDepth(t);
    float fMax = GetAreaMaximum(t,bluechannel);
    if (fMax > 0) fMax = t.z + fDistanceUnit;
    t = float4(t.x, fDepth, fMax, 0);
}
```

Figure 5: Individual channels (RGB) of the shadow map after preprocessing – original depth in red, nearest occluders in green, inversed distance to hard shadow boundary in blue (modified to be visible).

Final rendering

The final rendering step is straightforward. First we decide to which region (see Figure 1a) our point belongs. We sample the shadow map to decide whether the point lies in shadow or not. We also obtain the nearest occluder depth from the green channel, as well as the distance to the nearest shadow boundary from the blue channel.

If our point is in shadow we use the original depth value in the shadow map (red channel) to compute the disk radius. If this radius exceeds the maximum allowed radius, we clamp it.

If our point is lit we use the depth of the nearest occluder (green channel). If this value is 1.0, we consider the point as fully lit. Otherwise, we use the nearest occluder depth to compute the disk radius and clamp it to the maximum allowed radius if necessary.

Before doing any further processing we determine the distance of the point to the nearest shadow boundary by using the blue channel of the shadow map. If this distance is greater than the disk radius, the point cannot be in a penumbra region and we consider the point as either fully lit, or fully shadowed, and so are done. For all other pixels we compute the soft shadow values using stochastic sampled PCF as shown in the pseudo-code listing on the next page. For each iteration of the `while` loop we sequentially sample the rows of the noise map, and scale and translate the values in the red and green channels of the noise map to fit into the disk radius. We use the resulting locations as our PCF samples. The quality of the final shadow is greatly affected by the way in which we choose the initial sampling coordinates `vNoiseCoords` for the current pixel. If we just use interpolated coordinates (e.g. screen coordinates or shadow map coordinates) we get vertical banding artifacts. This happens because neighboring pixels that share the same *y* coordinate will use very similar PCF samples, due to the sequential reading in the `while` loop. To avoid this problem we store pure random values into the blue and alpha channels of the noise map. We sample the noise map using the canonical screen coordinates of our current pixel to index into the noise map's blue and alpha channels. This ensures that neighboring screen space pixels will not use the same PCF samples.

The following bit of pseudo-code shows the final rendering pass:

```
For each screen pixel p
{
    //Obtain projective shadow map coordinates.
    float2 vShadowMapCoord = input.xy / input.w;
    float4 cShadow = tex2D(ShadowMap, vShadowMapCoord);
    float fHardShadow = GetHardShadow(p, cShadow.r);
    float fPenumbraRadius = 1.0f;
    if (fHardShadow == 0)
        fPenumbraRadius = min(fMaxRadius, GetPenumbraRadius(p, cShadow.r));
    else if (cShadow.g == 1.0f)
        return fHardShadow;
    else
        fPenumbraRadius = min(fMaxRadius, GetPenumbraRadius(p, cShadow.g));
    if (fPenumbraRadius > cShadow.b)
        return fHardShadow;

    //Soft shadows
    float2 vNoiseCoords = tex2D(NoiseMap, p.xy).ba;
    float fSampleCount = GetSampleCount(fPenumbraRadius);
    fSampleCount = min(fMaxSampleCount, fSampleCount);
    float fCounter = fSampleCount;
```

```
    floaf fFinalShadow = fHardShadow;
    while (fCounter > 0)
    {
        float2 vNoiseCoord2 = tex2D(NoiseMap, vNoiseCoords);
        vNoiseCoord2 = vNoiseCoord2 * fPenumbraRadius;
        cShadow = tex2D(ShadowMap, vShadowMapCoord + vNoiseCoord2);
        fFinalShadow += GetPCFShadow(p, cShadow.r);
        vNoiseCoords.x = vNoiseCoords.x + fOneRow;
        fCounter--;
    }
    return fFinalShadow / fSampleCount;
}
```

Modifications to the standard implementation

Implementation on ps_2_0 hardware

The pixel shader 2.0 specification imposes certain constraints to the implementation. The first one is the relatively small number of instructions that a shader is allowed have, which forces us to break the PCF loop into several passes. Luckily there is the possibility of using a method similar to deferred shading [Thibieroz03], which enables us to completely skip the vertex processing phase. In the first pass we render the following data to each pixel of a texture: the shadow map coordinates (two channels), and the normalized distance from the light (one channel). The last channel contains the disk radius and the result of the hard shadow packed together. In subsequent passes we perform just the loop part of the algorithm and accumulate the shadow intensity in the alpha channel of the framebuffer. The final pass performs full lighting computations and uses alpha blending to modulate each result with the corresponding shadow value.

Implementation without preprocessing

Our algorithm can be used without preprocessing the shadow map. The distance map and nearest occluder map are used solely for calculating the disk radius and doing early-out tests. Without them the disk radius calculation proceeds as follows. For pixels that are occluded, the disk radius *q* is approximated by considering the difference in lightspace depth between the current pixel and its occluding shadow map texel (which is equivalent to what we described in the standard implementation). If the current pixel is not occluded, we can set *q* to its minimum allowed value and use this. Doing so yields aesthetically pleasing results. However, a more correct way to calculate *q* for this case would be to find the maximum of the difference in depth between each sample in the disk to its (possible) occluder (*not* the minimum of the distances; remember we have rendered the back faces to the shadow map, not the front faces). This implies that we first use the disk (with radius set to the maximum) to find this maximum difference in depth. We then use this distance (which is *d2* in Figure 1b) to calculate the corrected disk radius, and then perform the actual PCF with this corrected radius.

Results

The following figures show shadows produced by our soft shadow algorithm. The brute-force ps_2_0 version runs at 15-20fps with a resolution of 1024x768 on a Radeon 9800 with 17 shadow map preprocessing passes and 5 complete final rendering passes (with 10 noise texture reads and 10 shadow map reads for each pass).

Notice how the penumbra regions are noisy. In typical game situations this noise is masked by the use of high-frequency diffuse textures, or bump maps.

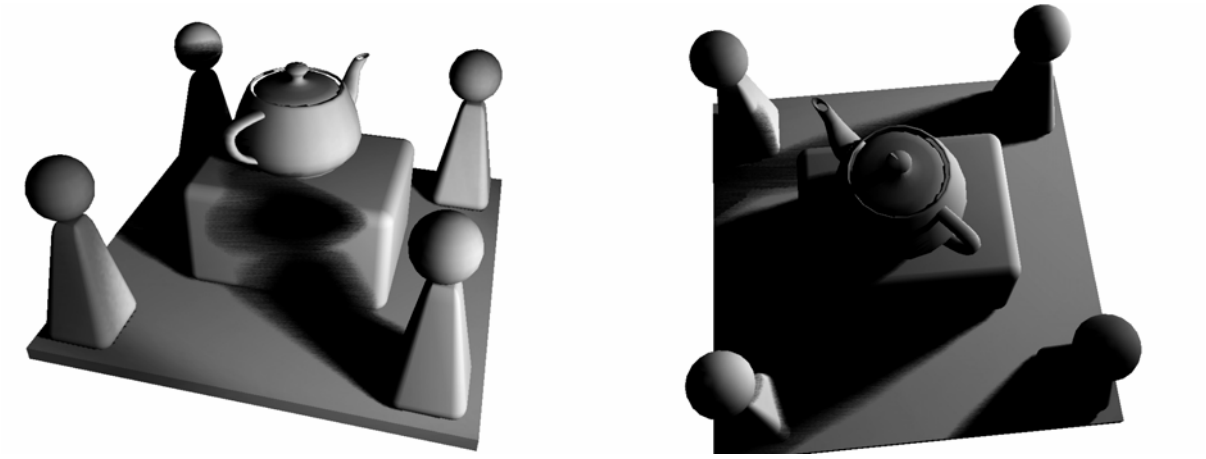

Figure 6: Final rendered images. These are using the current ps_2_0 version where the initial randomization step is skipped, so banding is visible.

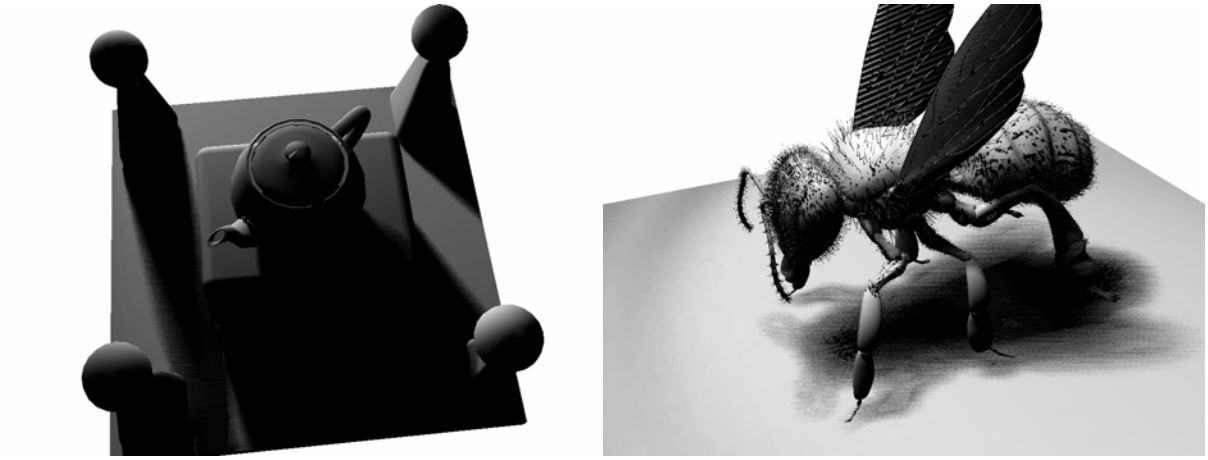

Figure 7: Final rendered images. These are from current ps_2_0 version.

Artifacts and limitations

Although our algorithm produces aesthetically pleasing results in most cases, it can also produce artifacts in the inner penumbra regions. These artifacts are quite common for shadow map based algorithms because the map stores the depth of only the nearest occluder. This means that if we try to compute the width of the penumbra region for objects that overlap in shadow map space we can get highly varying disk radii for points that are close together. Figure 8 shows this case on the right. For point *a* the disk radius is computed using the distance to occluder *occ2*. Point *b* should be deeper in shadow, but the shadow map stores *occ1* at this point and so our algorithm uses too large a disk radius for point *b*. Because of this mismatch, the shadow intensity can be vastly different from that of *a*, despite these two points being very close to each other.

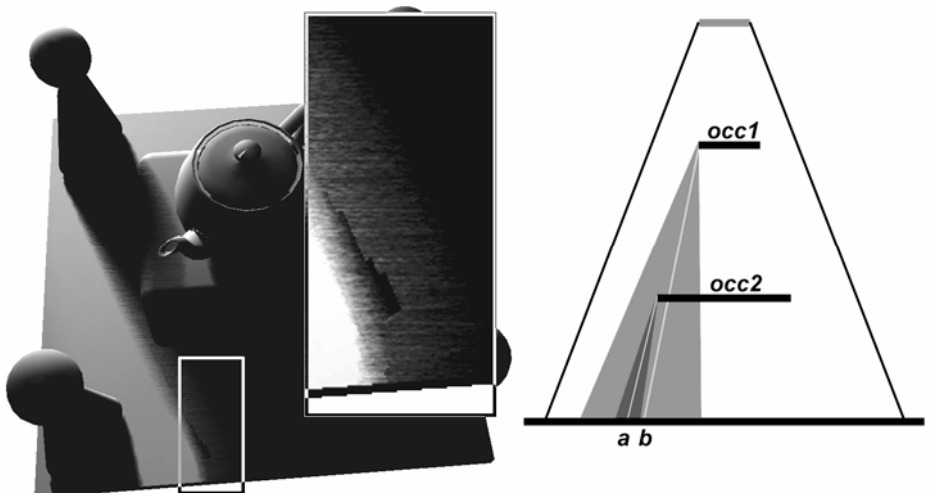

Figure 8: Incorrectly rendered penumbra region (left) and 2D illustration of the cause of the error (right). The brightness and contrast of the zoomed-in area was modified so the problem is more visible.

We also place the following limitations on the algorithm to make it behave nicely in terms of performance. The first limitation is that we define a maximum disk radius. Any greater radius is clamped to this maximum value. This allows us to limit the number of preprocessing steps. This limitation can be removed by using a different approach to preprocessing; see the Optimizations section.

The other limitation that we place on the algorithm is that we define a maximum number of PCF samples[4]. Because we can use only a finite number of samples, N, the eventual value PCF returns can ever be one of only N discrete values. This explains the banding artifacts in Figures 6 and 7, which we can mask by the use of noise (as explained in the previous section). In the limit as we take the number of samples to infinity, our method will produce smooth penumbra transitions, without any of the banding artifacts.

Optimizations

We could gain extra speed during the preprocessing phase by trying to change the preprocessing time from $O(n^2)$ to $O(n \log n)$, where n is pixel count of the shadow map, utilizing similar techniques used for separable filtering. We refer interested reader to the GDCE presentation by Chris Oat [Oat03]. By using this method we can cover a wider area using fewer passes.

We can also gain speed by using fewer PCF samples for points that are farther away from the viewer. Therefore we could modify Equation 2 to take into account the distance from the viewer.

We could also omit the noise map and store the Poisson disk samples in constant registers. Then we use the same set of random coordinates for each pixel (see the chapter “Poisson Shadow Blur” on page ?? for more on this subject).

[4] This is a limitation of the ps_3_0 specification, which can perform loops with only a non-variable loop-count.

Conclusion

We have presented a method that is capable of producing aesthetically correct soft shadows with inner and outer penumbrae. The actual implementation uses a modified version of PCF. The first modification is the variable width filtering kernel (i.e. disk). The second is the additional test performed during PCF that helps us to reject false shadow contributions. Using a Poisson disk distribution for the samples helps to hide banding artifacts caused by the limit on the number of samples. This banding is especially noticeable in large penumbra regions. The noise map can easily be replaced by a (smaller) one-dimensional texture or even by a fixed-size constant array of random coordinates for speed critical applications. With the additional power that flow control brings on ps_3_0 (or ps_2_x with caps) hardware, we are able to compute and use the minimum number of samples necessary for each pixel.

Acknowledgements

The first author would like to thank to Andrej Ferko for his valuable comments. The second author would like to thank Steve Hill for being such a good sounding board, and we would like to thank Eric Haines for his feedback and useful suggestions.

References

[Brabec02a] Brabec, S., and Seidel, H-P., "Single Sample Soft Shadows using Depth Maps", *Graphics Interface* (2002): pp. 219-228.

[Brabec02b] Brabec, S., Annen, T., and Seidel, H-P., " Practical shadow mapping", *Journal of Graphics Tools* (2002): pp. 9-18.

[Bunnell04] Bunnell, M., and Pellacini, F., "Shadow Map Antialiasing", in Randima Fernando, ed., *GPU Gems*, Addison-Wesley (2004): pp.185-192.

[Cook86] Cook, R., L., "Stochastic Sampling in Computer Graphics", *ACM Transactions on Graphics*, Vol 5, no.1 (1986): pp. 51-72.

[Hasenfratz03] Hasenfratz, J., M., Lapierre, M., Holzschuch, N., and Sillion, F., "A survey of Real-Time Soft Shadow Algorithms", *Computer Graphics Forum*, Vol 22, no.4 (December 2003).

[Oat03] Oat, Ch., "Real-Time 3D Scene Post-processing", GDCE 2003, available online at http://www.ati.com/developer/gdce/Oat-ScenePostprocessing.pps

[Reeves87] Reeves, W. T., Salesin, and D. H., Cook, R. L., "Rendering antialiased shadows with depth maps", *Computer Graphics*, (SIGGRAPH '87 Proceedings): pp 283–291.

[Thibieroz03] Thibieroz, N., "Deferred Shading with Multiple Render Targets", in Engel, W.F., ed., *ShaderX^2 – Shader Programming Tips & Tricks with DirectX9*, Wordware Publishing, 2003.

[Valient03] Valient, M., "Shadow mapping with Direct3D 9", in Engel, W. F., ed., *ShaderX^2 – Shader Introduction & Tutorials*, Wordware Publishing., 2003.

[Wang97] Wang, Y., and Molnar, S., "Second-Depth Shadow Maps", UNC-CS Technical Report TR94-019, 1994.

[Williams78] Williams L., "Casting Curved Shadows on Curved Surfaces", *Computer Graphics*, Vol 12, no.3 (SIGGRAPH '78 Proceedings): pp. 270-274.

[Zioma03] Zioma, R., "Reverse Extruded Shadow Volumes", in Engel, W. F., ed., *ShaderX^2 - Shader Programming Tips & Tricks with DirectX9*, Wolfgang F. Engel, ed., Wordware Publishing, 2003.