Whitepaper

Integrating Realistic Soft Shadows into Your Game Engine

Kevin Myers
Randima (Randy) Fernando
Louis Bavoil

Version 1.0 - February 2008

Integrating Realistic Soft Shadows Into Your Game Engine

Why Soft Shadows are Important

Despite the rapidly rising visual fidelity of today's games, few games have ever had shadows with penumbras that change size correctly based on occluder and receiver geometry as well as light size. Accurate soft shadows are one of the major steps games must take to become truly photorealistic.

In this whitepaper, we explain how to easily integrate an efficient method for creating realistic soft shadows on DirectX 10 and high-end DirectX 9 GPUs. Figure 1 shows our technique implemented in the recently released game *Hellgate: London*, developed by Flagship Studios.

Realistic Soft Shadows with PCSS **Ordinary Shadows**

Figure 1. Realistic Soft Shadows (Left) versus Ordinary Shadows (Right) in *Hellgate: London*

Image used with permission from Flagship Studios.

Three characters are visible in the screenshot, and each is positioned differently with respect to the ground. Realistic soft shadows help to give an accurate impression of each character's position. At the upper left is a zombie who is standing on the ground, and his shadow is correspondingly hard. On the right is a flying character with two glowing swords. Notice that the shadows of his feet are hard, since they are close to the ground. But in contrast, the shadows of his glowing swords are very soft. Finally, the main character (at the center) is jumping high off the ground, so his shadow is very soft. Altogether, the combination of shadows makes the image at left much more dynamic image than

the image the right. In addition, these visual cues can help players to discern more of what's happening on-screen, particularly when characters are jumping or objects are flying around.

In a more general sense, soft shadows are important because:

- **Realistic soft shadows help us to just spatial relationships between objects.** For example, the shadow of a bouncing ball will keep varying from hard to soft, and this allows us to perceive how high the ball is off the ground at any point in time.
- **Varying penumbras give shadows variety and character**. This is one reason why there is such a significant visual difference between real life and most computer games.

Most dynamic soft shadow techniques are typically very expensive and difficult to integrate into game engines. As a result, today's in-game shadows are either hard-edged or uniformly soft, where objects appear to float off the ground.

Implementation Overview

In this whitepaper, we describe how to easily integrate accurate soft shadows into a game engine, assuming only that the game already has a basic hard shadow implementation using shadow maps. The changes involve only **three simple steps**:

1. Use 32-bit floating point for the depth texture
2. Replace shadow map lookup with a new function call
3. Specify a value for the "light size" parameter

The goal of this whitepaper is to explain each of the steps in further detail.

The "new function call" that Step 2 refers to is an algorithm called Percentage-Closer Soft Shadows (PCSS). This algorithm was presented as a sketch during SIGGRAPH 2005, and is entirely encapsulated in one shader function call, which makes it an easy substitute for conventional shadow mapping. The algorithm has no external dependencies, so you don't have to understand how the algorithm works to use it. Still, we recommend that you learn a little about PCSS as background information. Please note, however, that the code embedded in this whitepaper is improved from the original version.

PCSS has these key characteristics:

- Generates perceptually accurate soft shadows
- Uses a single light source sample (one shadow map)
- Requires no pre-processing, post-processing, or additional geometry
- Seamlessly replaces a traditional shadow map query (embodies the same advantages as traditional shadow mapping—independent of scene

complexity, produced self-shadowing, works with alpha testing, displacement mapping, and so on)

- Runs in real-time on current consumer graphics hardware
- Easily scalable by varying the number of samples used

Step 1: Use a 32-Bit Depth Texture

Historically 24-Bit has been used for shadow mapping as this was the format that offered hardware-accelerated percentage-closer filtering (PCF). For PCSS though, we need to read the actual depth values in our blocker search. It is therefore advisable to take advantage of the extra precision offered by 32-bit as it is just as fast as 24-bit.

DirectX 10

For DirectX 10, there should be nothing you need to change. Just make sure you're using a shadow map with format `DXGI_FORMAT_R32_TYPELESS`. Then use `DXGI_FORMAT_D32_FLOAT` for the `DepthStencilView` and `DXGI_FORMAT_R32_FLOAT` for the `ShaderResourceView`.

DirectX 9

If you were already doing DirectX 9 hardware shadow mapping, your shadow map creation call probably looked something like this:

```
IDirect3DTexture9 *pShadowMapTexture;

IDirect3DSurface9 *pShadowMapSurface;

…

D3DDev->CreateTexture(
        texWidth,
        texHeight,
        1,
        D3DUSAGE_DEPTHSTENCIL,
        D3DFMT_D24S8,
        D3DPOOL_DEFAULT,
        & pShadowMapTexture);

pShadowMapTexture0-> GetSurfaceLevel( 0, &
pShadowMapSurface );

…
```

Then you would render all shadow casting geometry to the shadow map by setting `pShadowMapSurface` as the depth-stencil surface with color writes

disabled (since you are only writing depth). For PCSS what we really want is the 32-bit floating-point depth value, not the filtered result. This requires us to render to a `D3DFMT_R32F` render target with color writes enabled. To create such a texture we change our above `CreateTexture` call to:

```
D3DDev->CreateTexture(
        texWidth,
        texHeight,
        1,
        D3DUSAGE_DEPTHSTENCIL,
        D3DFMT_R32F,
        D3DPOOL_DEFAULT,
        & pShadowMapTexture);
```

Make Sure Your Shader Writes Depth to Color

Since your current shadow map generation pass should already be computing light space projected z you just need to make sure your pixel shader is writing this out (since we're now using the color computed by the pixel shader). Your vertex and pixel shader should look something like this:

```
void ShadowMapVS(
   in float3 inPos : POSITION,
   out float4 outPos : POSITION,
   out float depth : TEXCOORD )
{
   outPos = mul( inPos, WorldViewProjection );
   depth = outPos.z;
}

void ShadowMapPS( in float depth : TEXCOORD )
{
   return depth;
}
```

DirectX 10 Shader Changes

It's important that you use point-sampled filtering (for example, `D3D10_FILTER_MIN_MAG_MIP_POINT`) instead of comparison filtering (for example, `D3D10_FILTER_COMPARISON_MIN_MAG_LINEAR_MIP_POINT`) on your texture.

If you were using one of the comparison modes to get hardware-accelerated PCF, you need to use point sampling so that you can access the discrete depth values and make calculations with them.

If you were already using point-sampled filtering, and were manually doing comparisons in the shader, you will be replacing the manual comparisons with the PCSS shader in Step 2.

DirectX 9 Shader Changes

After Step 1, your original shadow shader will no longer generate a correct image, because the shadow map texture lookup now returns a floating-point value instead of a gray-scale filtered color. But don't worry – in Step 2, you'll easily be able to check if you did Step 1 correctly.

If you really want to check Step 1, an easy way is to replace your original `tex2D()` call with a quick comparison.

Before:

```
// coords.xy contains the u-v coordinates to look up
// coords.z is the depth value to compare against

shadowMap = tex2D(shadowMap, coords);
```

After:

```
if (tex2D(shadowMap, coords.xy) > coords.z )
   // shadowed case
   shadowColor = float4(0.0f, 0.0f, 0.0f, 0.0f);
else
   // lit case
   shadowColor = float4(1.0f, 1.0f, 1.0f, 1.0f);
```

Step 2: Replace Shadow Map Lookup with PCSS Function

Now that you have a floating-point texture, you have only to replace the old shadow map texture lookup with the PCSS function call:

DirectX 10

Before:

```
// Rough example
Sampler PCF_Sampler
{
   Filter = D3D10_FILTER_COMPARISON_MIN_MAG_MIP_LINEAR;
   AddressU = Clamp;
   AddressV = Clamp;
};

shadowColor = shadowMap.SampleCmpLevelZero(
PCF_Sampler, coords );
```

After:

```
shadowColor = PCSS( shadowMap, coords );
```

DirectX 9

Before:

```
shadowColor = tex2D( shadowMap, coords );
```

After:

```
shadowColor = PCSS( shadowMap, coords );
```

If you're lucky (meaning that your game's units match the default settings for the PCSS shader), you'll already see accurate soft shadows when you run the game! In practice, however, you'll probably have to tune the light size, as explained in Step 3.

Unless everything is magically working perfectly, it's a good idea to do a few sanity checks at this point:

First, replace the "return PCF_Filter;" line of the PCSS shader with:

```
return PCF_Filter( shadowMapTex, coords, 0 );
```

This should generate hard shadows.

Also, you can try:

```
return PCF_Filter( shadowMapTex, coords, 1 );
```

This should generate uniformly soft shadows.

If both of these tests work, you can move to Step 3.

Step 3: Tuning the Light Size

The light size allows you to control the overall softness of the scene's shadows. Larger lights will of course result in softer shadows. This parameter is something you can change with artistic preference – unless the light size is so large that the algorithm starts filtering outside the shadow map, any light size value will produce a realistic image.

Performance

The PCSS shader has two parameters that you can tune for performance:

- `BLOCKER_SEARCH_NUM_SAMPLES`. This specifies how many samples are used for the blocker search step. Multiple samples are required to avoid holes in the penumbra due to missing blockers.

- `PCF_NUM_SAMPLES`**.** This specifies how many filtering samples are used per dimension. The more samples the smoother are the shadows.

A value of 4 (16 samples) is common for the blocker search as well as the PCF filtering, and is equal to what many shipping games do today. Higher values naturally produce higher quality, but reduce performance. If you have detailed textures, they will help to mark any undersampling artifacts, allowing you to keep the number of filtering samples low.

If you change the values, you must also generate other Poisson disks that equal the number of samples you want. For more on Poisson disks and some source code for generating them see this web page:
http://www.cs.virginia.edu/~gfx/pubs/antimony/

Performance is very high on recent GPU such as GeForce 8800 GTX or GeForce 7800 GTX. Even at 1600 x 1200, with 25 search samples and 16 PCF samples, the technique runs at well over 100 fps (with all pixels on screen shaded). In practice, you can probably restrict the number of pixels shaded to get even better performance.

Integration in Game Settings Menu

The PCSS technique is ideally suited for GeForce 8 Series and high-end GeForce 7 Series GPUs that have powerful pixel shaders and texturing subsystems. Therefore, the best way to expose PCSS to people playing your game is to have an option in your Game Settings called "High Quality Soft Shadows". If you already auto-detect graphics card performance, the technique can probably be enabled for the highest quality modes.

Ideas for Improvement

To further improve image quality and performance, you can try the following:

- **Irregular filtering.** By spacing the filtering samples irregularly, you can reduce banding without unduly increasing the number of samples. In the code below we use a 16 tap poisson disk (points evenly and randomly distributed about the unit circle) to do the blocker search.
- **Use SAVSM.** The filtering step is very costly (in the code below we do 256 lookups) making PCSS a perfect candidate for Summed Area Variance Shadow Mapping (see Andrew Lauritzen's chapter in *GPU Gems 3*). This technique allows for filter kernels of any size to change per pixel.
- **Use a variable depth bias inside the PCF kernel.** Normally, PCF compares the surface depth (coord.z) with the depths in the kernel. For small

kernel sizes, self-shadowing can be handled by biasing coord.z with a combination of a slope-based bias and a constant bias. As the kernel size increases, larger and larger depth biases need to be used to avoid false self-shadowing. Doing a proper bias is important to avoid false occluders to be considered during the blocker search as well as the PCF filtering. A solution is to offset coord.z for every sample in the direction of the local depth gradient (dz/du, dz/dv), by a magnitude increasing with the 2D distance between the sampled uv and the center uv. For more details see [Schuler2006] "Multisampling Extension for Gradient Shadow Maps".

Conclusion

We hope this whitepaper helps you to improve the shadow quality in your next game. PCSS provides an accessible way to create realistic soft shadows with relatively low cost, allowing you to add an unprecedented level of immersiveness to your game.

PCSS Shader Code

```
#define BLOCKER_SEARCH_NUM_SAMPLES 16
#define PCF_NUM_SAMPLES 16
#define NEAR_PLANE 9.5
#define LIGHT_WORLD_SIZE .5
#define LIGHT_FRUSTUM_WIDTH 3.75

// Assuming that LIGHT_FRUSTUM_WIDTH == LIGHT_FRUSTUM_HEIGHT
#define LIGHT_SIZE_UV (LIGHT_WORLD_SIZE / LIGHT_FRUSTUM_WIDTH)

Texture2D<float> tDepthMap;

cbuffer POISSON_DISKS
{
        float2 poissonDisk[16] = {
                float2( -0.94201624, -0.39906216 ),
                float2( 0.94558609, -0.76890725 ),
                float2( -0.094184101, -0.92938870 ),
                float2( 0.34495938, 0.29387760 ),
                float2( -0.91588581, 0.45771432 ),
                float2( -0.81544232, -0.87912464 ),
                float2( -0.38277543, 0.27676845 ),
                float2( 0.97484398, 0.75648379 ),
                float2( 0.44323325, -0.97511554 ),
                float2( 0.53742981, -0.47373420 ),
                float2( -0.26496911, -0.41893023 ),
                float2( 0.79197514, 0.19090188 ),
                float2( -0.24188840, 0.99706507 ),
                float2( -0.81409955, 0.91437590 ),
                float2( 0.19984126, 0.78641367 ),
                float2( 0.14383161, -0.14100790 )
};
};
```

```
float PenumbraSize(float zReceiver, float zBlocker) //Parallel plane estimation
{
    return (zReceiver - zBlocker) / zBlocker;
}

void FindBlocker(out float avgBlockerDepth,
                 out float numBlockers,
                 float2 uv, float zReceiver )
{
    //This uses similar triangles to compute what
    //area of the shadow map we should search
    float searchWidth = LIGHT_SIZE_UV * (zReceiver - NEAR_PLANE) / zReceiver;

    float blockerSum = 0;
    numBlockers = 0;

    for( int i = 0; i < BLOCKER_SEARCH_NUM_SAMPLES; ++i )
    {
       float shadowMapDepth = tDepthMap.SampleLevel(
            PointSampler,
            uv + poissonDisk[i] * searchWidth,
            0);
        if ( shadowMapDepth < zReceiver ) {
                blockerSum += shadowMapDepth;
                numBlockers++;
            }
     }

    avgBlockerDepth = blockerSum / numBlockers;
}

float PCF_Filter( float2 uv, float zReceiver, float filterRadiusUV )
{
    float sum = 0.0f;
    for ( int i = 0; i < PCF_NUM_SAMPLES; ++i )
    {
        float2 offset = poissonDisk[i] * filterRadiusUV;
        sum += tDepthMap.SampleCmpLevelZero(PCF_Sampler, uv + offset, zReceiver);
    }
    return sum / PCF_NUM_SAMPLES;
}

float PCSS ( Texture2D shadowMapTex, float4 coords  )
{
    float2 uv = coords.xy;
    float zReceiver = coords.z; // Assumed to be eye-space z in this code

    // STEP 1: blocker search
    float avgBlockerDepth = 0;
    float numBlockers = 0;
    FindBlocker( avgBlockerDepth, numBlockers, uv, zReceiver );

    if( numBlockers < 1 )
    //There are no occluders so early out (this saves filtering)
       return 1.0f;

    // STEP 2: penumbra size
    float penumbraRatio = PenumbraSize(zReceiver, avgBlockerDepth);
    float filterRadiusUV = penumbraRatio * LIGHT_SIZE_UV * NEAR_PLANE / coords.z;

    // STEP 3: filtering
    return PCF_Filter( uv, zReceiver, filterRadiusUV );
}
```