TNCG14
Soft Shadows in real time by Imperfect Shadow Maps and Screen Space Ambient Occlusion

Johan Beck-Norén, johbe559@student.liu.se
Andreas Valter, andva287@student.liu.se

September 9, 2013

Abstract

This report describes a method for simulating low frequency soft shadows in a scene by creating shadow maps from a point representation, and screen space ambient occlusion to take care of high frequency shadows. This makes it possible to increase the light source count in the scene and make the lighting converge towards a real representation of area lights while still running in real time.

1 Background and prior work

Soft shadows is the result of area lights hitting the object from different angles, creating a shadow with two stages, *umbra* (2 in figure 1) and *penumbra* (1 in figure 1). Umbra describes the part that is fully in shadow, the penumbra region is partially hidden from the light. The penumbra part is only modeled when we deal with soft shadows. This effect is something that is important in computer graphics for creating a believable scene. For offline rendering techniques like ray tracing it is achieved without a lot of extra work. But when dealing with real time rendering, it is expensive to achieve something that is even remotely close to reality.

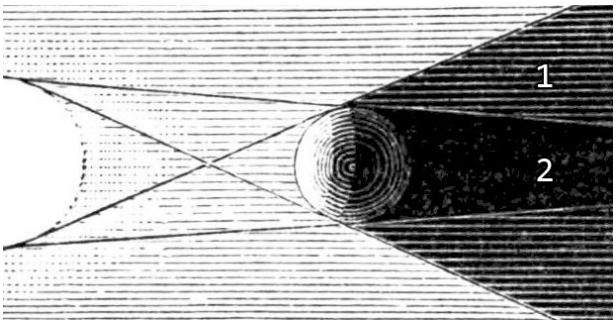


Figure 1: Showing shadow behavior, 1 is the region called penumbra and 2 is called umbra. Image from [3].

Hard shadow approaches only simulate the umbra. The simplest approach to this would be planar shadows which is projection of objects onto other objects. This approach creates hard shadows between objects, but the technique does not support self shadowing and also requires a planar surface for the shadow.

A more advanced idea is to render depth maps from the lights and projecting them on the scene, this technique is called *Shadow Mapping*. By comparing the length between the light source and the point it is possible to decide if each point is visible from the light source or not. But this technique still does not provide a soft shadow, at least not without modifications.

2 Imperfect shadow mapping

The ideal would be to create a lot of shadow maps from each light source to simulate area lights as close as possible. But due to the fact that shadow maps are quite expensive to create, it is impossible to do this without modification with the hardware that is available at the moment. The idea behind the Imperfect shadow mapping algorithm is to create a lot of low resolution shadow maps, using a simple point representation of the scene, thus allowing for fast rendering of complex scenes.

2.1 Scene Point Representation and Imperfect Shadow Maps

To reduce the computational load we simplify the scene. We do this by approximating selected triangles in the scene with points of roughly uniform density, as in [2]. Triangles are chosen randomly with probability weighted against the triangle's surface area. Shadow maps are then created from this point representation. For each virtual point light (VPL), the points are splatted into the depth buffer, using a geometry shader, by a factor proportional to the distance squared between the VPL and the points representing the scene.

Since we are using a point representation of the scene the depth maps generated from the VPLs often contain gaps. This is remedied by using a pull-push interpolation algorithm [1] to fill in some of the gaps. The algorithm consists of a pull phase and a push phase. Using an image pyramid in bottom-up order, the pull phase reduces the depth map's resolution by a factor och 2 for each step. The downsampling has a depth threshold condition for deciding which pixels are to be used to average the pixel on the coarser level (Figure 2).

The push phase works with the built image pyramid in top-down order, from coarser to finer levels. In this phase, only pixels *outside* a set depth threshold compared to the corresponding pixel in the coarser pyramid level are considered. The push phase interpolation scheme can be seen in figure 3.

When used for lighting computations, the shadow map needs to cover a full hemisphere of depth information.

2.2 Screen Space Ambient Occlusion

Screen-Space Ambient Occlusion (SSAO) is a rendering technique for approximating the ambient occlusion in a computer graphics scene in real time. It works under the assumption that points in the scene which are close to other points will

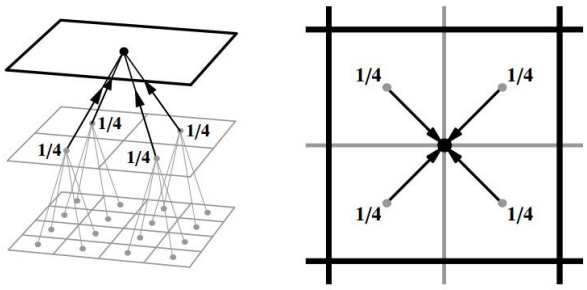


Figure 2: Pull-phase interpolation. Image from [1].

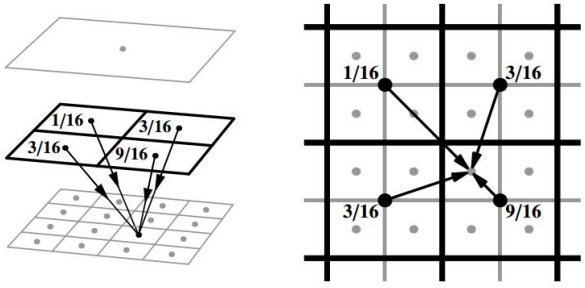


Figure 3: Push-phase interpolation. Image from [1].

be more occluded from the ambient light present in the scene. The first implementation of this technique was used in the game Crysis released in 2007 (Figure 4). The implementation used the depth buffer to sample the depth for a pixel, and then sample the depth of surrounding points within a hemisphere in three dimensions oriented along the pixel's normal vector. From these sample points, the pixel's occlusion factor would be calculated and later subtracted from the ambient portion of the scene's final lighting.

3 Implementation

3.1 Virtual Point Lights

Virtual point lights (VPLs) are created inside the light source. In our implementation we create them in a defined area above the scene to simulate sunlight. It is also possible to spawn them with environment mapping using a sky dome and extracting light sources from it. The simplified

Figure 4: SSAO as first implemented by Crytek in 2007.

point representation of the scene is then rendered from each of these positions using a full hemisphere for depth information. This is done by changing the field of view in the projection from each of the light sources.

3.2 Scene Point Representation and Imperfect Shadow Maps

For the point representation we randomly choose a triangle with probability based on the area of each triangle, making it more plausible that a larger one is picked, to represent with a point. The points on each selected triangle is generated with uniformly distributed homogeneous barycentric coordinates with the pseudo code in Listing 1. A geometry shader is then used to convert each point to a quad, with the same normal as the triangle that the point was generated from. This quad is then rendered as a circle in the fragment shader. By doing this, the points are generated quickly in parallel on the GPU.

Listing 1: Code for finding random point on triangle.

```
t1 = random(0, 1)
t2 = random(0, 1)
if t1 + t2 > 1
        t1 = 1.0 - t1
        t2 = 1.0 - t2
t3 = 1.0 - t2 - t3
position = vert1 * t1 +
        vert2 * t2 + vert3 * t3
```

We create many low resolution shadow maps, one for every VPL. These low-resolution shadow maps are stored in a single high-resolution texture. Since we use a sparse point representation to approximate the scene, there will be gaps and missing parts in the depth maps created. To remedy this we use a pull-push interpolation technique [1] to fill in these gaps. The pull-push interpolation is implemented in shaders and is applied after creating the point representation and before projecting the shadow maps on the scene. After the projection, all shadow maps are stored in an array of 2 dimensional textures. We then send in all shadow maps, together with the transformation matrix for each one of them to the shader responsible for merging them together and calculating the shadow value. For large amounts of shadow maps, this creates a problem because shaders have a maximum number of allowed uniform input variables. To account for this, the transformation matrices are transferred to the GPU as uniform buffers which allows for a larger amount of available uniform data which allows for enough shadow maps to be used.

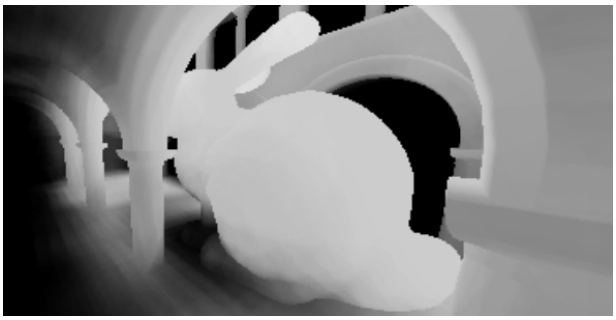
3.3 Screen Space Ambient Occlusion

In the implementation of SSAO we store the scene's depth buffer, per-pixel normals, and per-pixel positions as textures. The textures are downsampled to half their original resolution. This is in part to save memory and computation time, and in part because the SSAO generally results in some noise and artifacts. If the textures were not downsampled, we would have to apply a blur to the resulting SSAO to remedy this. We still apply a blur, but with a smaller filter kernel than would be needed for full sized textures.

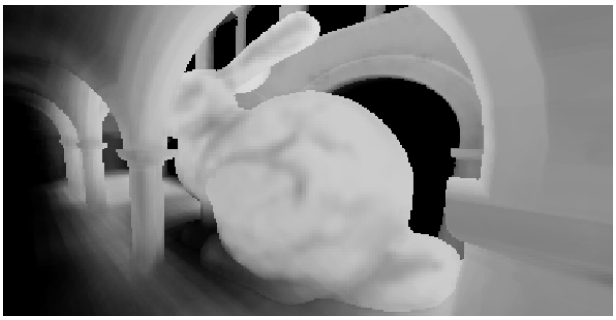
We generate a 64x64 texture of random values in the interval $[-1, 1]$ on the CPU and tile it over

the depth/normal/position textures and use this to randomly sample depth values for a pixel. The usage of textures in this way results in that we do not perform a sampling within a hemisphere in 3D, as the original implementation, but instead use a deferred approach to the technique.

Artist-controllable variables include sampling radius, sampling cone angle, intensity, and attenuation. All calculations are done in parallel on the GPU using shaders.



(a) No SSAO



(b) With SSAO

Figure 5: Comparison showing the contribution of our SSAO to a scene's ambient light.

3.4 Multicore

Throughout the implementation, we are taking advantage of both the full power of the CPU and GPU. During the initialization part, we use OpenMP for generating points on triangles. After the initialization process, we perform all heavy computations on the GPU. This means that we are using the power of the graphics card to render our scene in an highly optimized way.

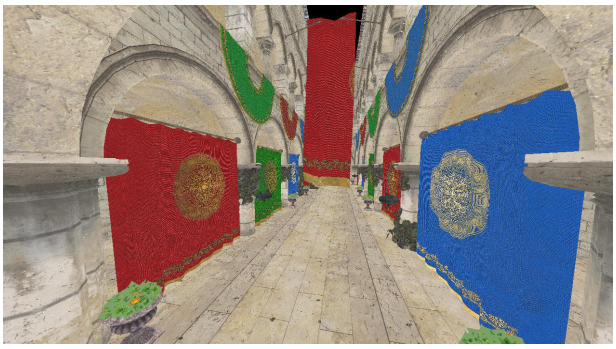
4 Results

Figure 6 shows three comparison shots. Figure 6(a) shows the scene as-is, with textures and no lighting applied. Figure 6(b) shows our implementation of soft shadows applied using 105 VPLs. The bottom figure 6(c) shows the same scene as figure 6(b) but with SSAO added. Notice the detail of the creases in the hanging cloths and the added dark areas around the pillars compared to figure 6(b).

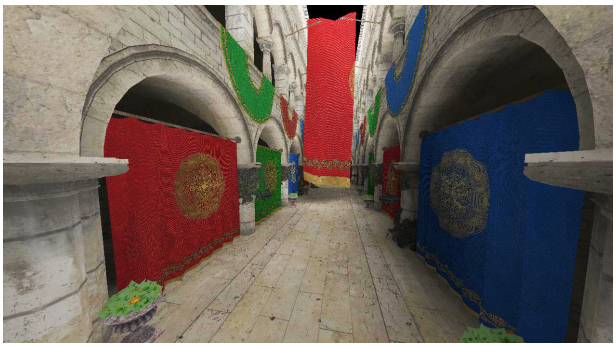
The scene was run on a computer with an Intel Q6600 CPU, 4gb RAM, and a NVidia GeForce 560Ti graphics card, and ran at around 25 frames per second at a resolution of 1280x720 pixels.

5 Discussion

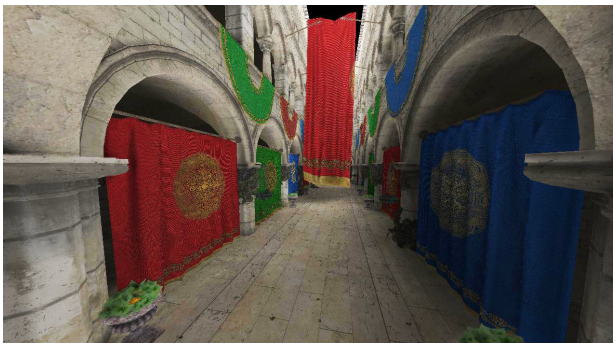
As is shown in Figure 6, the Imperfect Shadow Mapping algorithm is a good way to simulate soft shadows in real time. The problems presented in the original report with having high detail shadows were also seen in our simulation. The implementation of Screen space ambient occlusion, shown in Figure 6(c) seems like a good way to increase the lost details for the shadow. It offered a good and cheap way to increase the over all quality of the shadows and make the simulation more believable.



(a) No lighting



(b) Soft shadows



(c) Soft shadows with SSAO

Figure 6: Comparison figures showing a scene with (a) no lighting, (b) our soft shadows implementation, and (c) soft shadows with SSAO.

References

[1] R. Marroquim, M. Kraus, and P. R. Cavalcanti. Efficient point-based rendering using image reconstruction. pages 101–108, 2007.

[2] T. Ritschel, T. Grosch, M. H. Kim, H.-P. Seidel, C. Dachsbacher, and J. Kautz. Imperfect Shadow Maps for Efficient Computation of Indirect Illumination. *ACM Trans. Graph. (Proc. of SIGGRAPH ASIA 2008)*, 27(5), 2008.

[3] W. J. Youmans. Images and shadows. *Popular Science Monthly*, 4:671, 1874.