Software Shadows For Ray Tracing Using Hardware Cameras

Kyle Moore
kjmoore@cs.unc.edu

Abstract

The goal of ray tracing is to create images that are as accurate to real life as possible. Unfortunately, attaining this goal typically requires an extremely large amount of processing time. In this paper I propose a method for creating accurate soft shadows using graphics hardware to accelerate the process.

1. Introduction

I came up with this approach while working on the soft shadow algorithm of my ray tracer. The soft shadows my ray tracer was originally producing had noticeable pattern effects. Although jittering and anti-aliasing helped somewhat, the patterns persisted. The only way I found to fix this problem was to increase the number of shadow rays per sample. However, as the number of shadow rays increased, the render times increased exponentially. I was not satisfied with this as render times were already quite high.

I began to consider alternate approaches to my current method, and developed this new approach in the process. This new approach attempts to use the GPU to simulate shooting many shadow rays at once. Offloading some of the work to the GPU also allows for performance gains due to parallelization.

2. Previous Works

In this section, I cover some previous approaches to this problem that are similar to my approach.

2.1 Cone Tracing

In the early days of computer graphics when ray tracing was the state of the art, Amanatides [1] realized that shooting a single line ray was a pretty poor approximation of how light behaved in nature. He proposed a technique where line rays are replaced with cone-shaped rays. This creates areas of intersection as opposed to point intersections. These areas are more accurate than point samples and effectively give you soft shadows, anti-aliasing, and numerous desired effects.

The drawback, however, is that the cone intersections are very difficult to calculate and require a significant amount of computation. For this reason, this technique never became popular. Instead, distributed ray tracing became the norm as it produced images of equal quality with much less complexity. My approach shares some ideas with this paper but ultimately is quite different.

2.2 Geometry-based Soft Shadow Volumes

The second paper I looked at [2] sounded like it might have a similar approach to mine, as I wanted to use graphics hardware to create shadows. As I read further, I found that their approach was a forward approach, where shadow silhouettes are found and polygons are used to cast shadows onto an object. This method was designed for modern graphics pipelines. My approach is a backwards approach, so I won't go into any more detail about their technique.

2.3 Soft shadow volumes for ray tracing

The technique described in [3] tries to solve the same problems I hope to solve using similar methods. Their approach samples the light source from the point to be shaded. However, the amount of light occluded is not calculated by sampling the light with a hardware camera as in my approach. Instead, occluder silhouettes are found and the amount of occlusion is determined by computing areas based on these silhouettes. My technique simplifies this by removing silhouette detection and letting the hardware approximate the areas of occlusion directly with sampling.

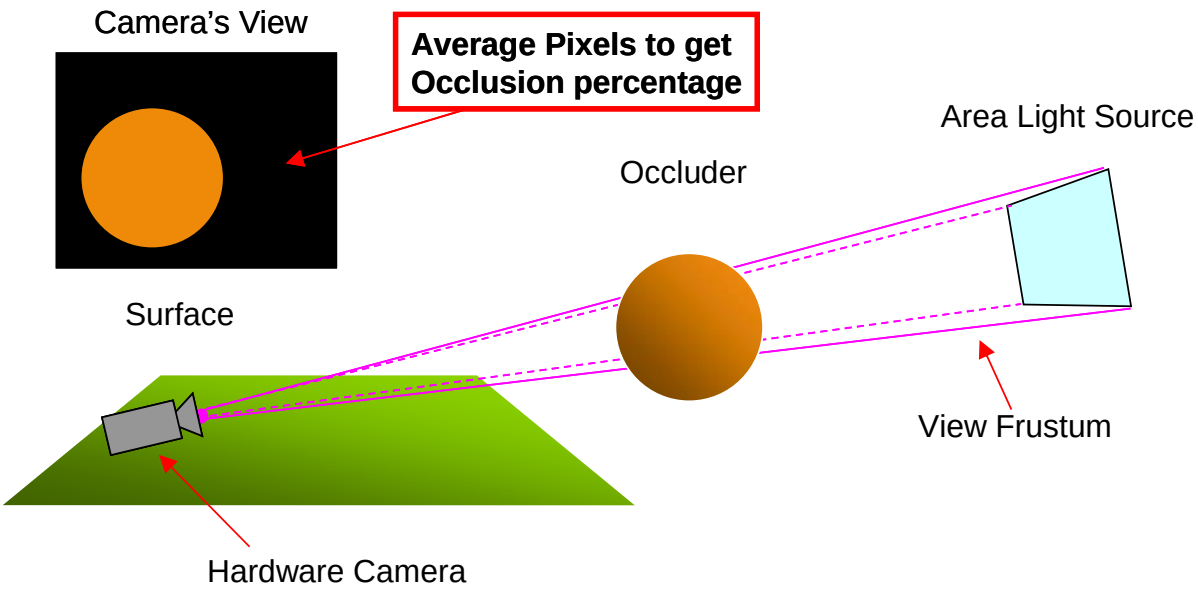

Figure 1: Using hardware cameras to calculate light occlusion

3. Approach

My approach is quite simple. First, a proxy of the scene is created. This proxy scene need not have any colors or material properties, only geometry. Also, the geometry in this proxy scene only needs to be approximate – high polygon spheres won't improve image quality greatly as the general shape matters most (the number of rasterized pixels in a low-poly sphere and a high-poly sphere are not very different).

Next, the ray tracer is to be run normally, except for when it comes to casting shadow rays. Instead of casting a bunch of shadow rays, a hardware camera is positioned at the point where the shadow rays would be emanating from. It is aimed towards the light that is currently being sampled, and the camera frustum is set such that the far clipping plane of the frustum is the same shape as the light source. It is assumed in this approach that all light sources are quads.

The scene is then rendered to a viewport. The size of the viewport determines how accurately the light is sampled. Essentially the pixels in the hardware render act as a group of shadow rays. Because of this it is possible to simulate a large number of shadow rays efficiently. In my implementation, a viewport size of 32x32 seemed to be the upper limit of quality. Going higher created no noticeable increase in image quality.

Once the render is complete, a count of the number of pixels that got rasterized is found and divided by the viewport size to yield the total light occlusion percentage. The NVOcclusionQuery makes this task simpler. Figure 1 depicts the entire process.

4. Results

To test this approach I compared the image quality and render times of my ray tracer before and after the addition of the new technique. Render times can be seen in Table 1. My first implementation, which I'm calling "hardware camera naïve", is employing the hardware render whenever an intersection is detected. "Camera w/ optimizations" means that a few optimizations were put into place to prevent unnecessary renderings from being done, such as when the point

# of samples	Shadow rays	Hardware camera naive	Camera w/ Optimizations
4x4	42 secs	151 secs	
17x17	152 secs	158 secs	
32x32	447 secs	160 secs	96 secs
64x64		181 secs	
128x128		241 secs	

Table 1: Render times

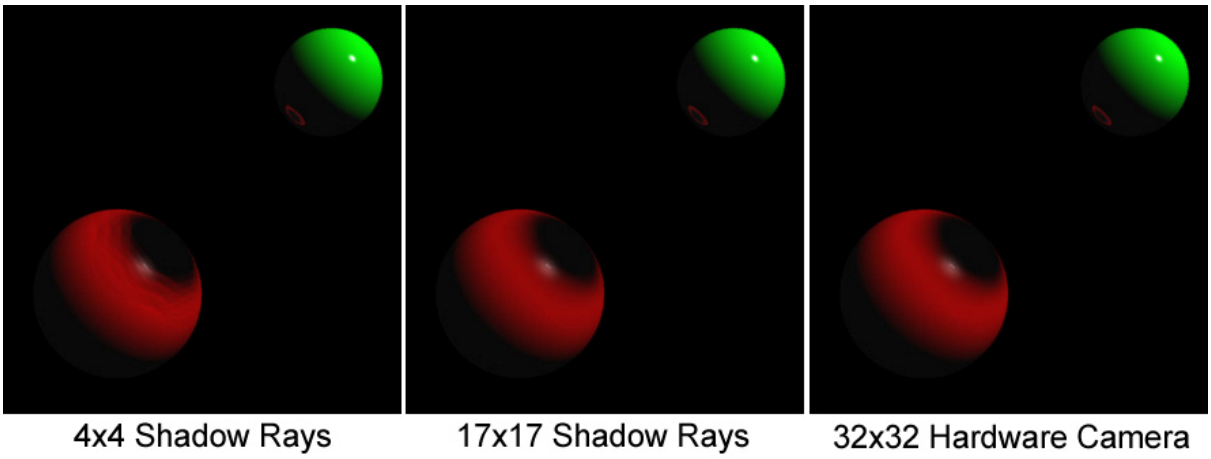

Figure 2: Quality comparison

is in complete shadows or no shadows.

As can be seen in Table 1, this approach can help improve render quality while decreasing render times. The 32x32 camera render took 96 seconds, where as the 17x17 software shadow rays render took 152 seconds and was not as accurate. Figure 2 comparatively show the renders.

5. Future Work

The current implementation of this approach does not reach the full potential of parallelization between the CPU and the GPU. Because of this, the graphics pipeline is constantly starved. I would like to improve my ray tracer such that it can be working on multiple pixels simultaneously. This would help keep the graphics hardware running at full potential.

Using NVOcclusionQuery, one could conceive of a method to handle any shape light using this technique. The modifications to the current approach would be simple. First, one would need to modify the camera frustum to ensure the geometric model of a light could fit in it. Second, using NVOcclusionQuery, render a model of the light to get the total number of pixels the light can possibly occupy. Then render the scene and render the light again with NVOcclusionQuery. The new pixel count divided by the original count is the percentage of the light seen by the spot being sampled. In this way, a light source of any shape could be handled.

Another idea that came to me while I was working on this project was to try a similar approach for global illumination. Since global illumination occurs from diffuse light bounces – which are inherently fuzzy – a basic hardware rendering of the scene could be sufficient to calculate believable global illumination at a low cost.

6. Conclusions

Although the results of this project are nothing spectacular, I believe it serves well as a proof of concept. It may be a lost cause from the start, since programmable GPUs already facilitate ray tracer quality rendering at interactive frame rates. Nevertheless, it was interesting to try to augment a software ray tracer with graphics hardware. Overall, I would consider this project successful.

7. Acknowledgements

I would like to thank Anselmo Lastra for his guidance and for suggesting the use of the NVOcclusionQuery.

8. References

[1] J. Amanatides. “Ray tracing with cones.” In *Proceedings of the 11th Annual Conference on Computer Graphics and interactive Techniques* H. Christiansen, Ed. SIGGRAPH '84. ACM Press, New York, NY, pp. 129-135, 1984.

[2] U. Assarsson and T. Akenine-Möller. “A geometry-based soft shadow volume algorithm using graphics hardware.” ACM Trans. Graph. 22, 3 pp. 511-520, Jul. 2003.

[3] S. Laine, T. Aila, U. Assarsson, J. Lehtinen and T. Akenine-Möller. “Soft shadow volumes for ray tracing.” ACM Trans. Graph. 24, 3, pp. 1156-1165, Jul. 2005.