Building a Fast Ray Tracer

Andrew Leing and Gerrett Diamond

May 1, 2014

Abstract

Ray tracing is often used in renderers, as it can create very high quality images at the expense of run time. It is useful because of its ability to solve many different problems in image rendering. Unless built efficiently, ray tracers take extreme lengths of time to complete their task. Optimizations are needed in order to keep the rendering time feasible.

This paper focuses on a few optimization choices in order to drastically decrease the render times. The parts of an image our renderer focuses on are super-sampling antialiasing and soft shadows, while the renderer itself is geared for maximum speed. One more thing our paper details is the parallelization of ray tracing and how it can be easily taken advantage of in order to maximize efficiency.

1 Introduction:

Creating realistic renderings is a huge focus in graphics, and there are multiple techniques to create high quality images for various applications like movies or picture generation. Ray tracing is one such method providing an easily understood base algorithm for rendering. The benefits of ray tracing is its versatility in the many ways it can be implemented to represent a variety of image effects, although when built naively can take a massive length of time to render. Problems ray tracing can solve go from fixing aliasing, to simulating refraction, and properly representing motion blur. We wanted to look into a few of the rendering problems ray tracing can solve while focusing on optimizing our renderer to make it as fast as possible.

The rendering problems our implementation looks to solve are anti-aliasing and producing soft shadows. Aliasing is the issue where a ray traced image produces jagged edges along the borders of objects or other boundaries. Anti-aliasing is the method of smoothing out those jagged edges, where the pixel is sampled in some way to account for the edge going through it. We examine the solution of super-sampling the pixel to solve aliasing, though only when necessary. Soft shadows are the solution to producing more realistic shadows from an object, where a naive ray tracer will create a circular shadow of just the umbra. Our soft shadow technique adds in and blends the penumbra of the object shadow. This paper goes into the details of the methods we used to solve those problems and the optimizations we made to increase the speed of those solutions. We also take advantage of the way rays are calculated to parallelize our renderer as another way of strictly improving the speed. This can be done easily because of the independence of the ray traces per pixel.

2 Related Work

Realistic looking generated images can be made through high quality rendering techniques. A well-known one is ray tracing [2] where the scene is sampled through the lens of the eye point viewing the area. Rays are cast from the eye through the lens into the scene to intersect with objects and sample the amount of light in that spot. Ray tracing can generate quality pictures, but requires large amounts of sampling to gather to correct shading and colors of a scene. Sampling can become computationally extremely expensive, so

techniques are researched to make sampling a feasible solution while maintaining a high quality image. One method of heavy sampling with higher efficiency is to adaptively sample the image [1]. Adaptive sampling still heavily samples the image, but only in pixels that are determined to be problematic. Using such a method allows for a ray tracer to get ever closer to real time rendering speeds.

Real time ray tracing comes in many steps as detailed in Real time Ray Tracing [5]. They talk about three different places where it can be achieved and under different means. Although we chose to implement our faster ray tracer on a single machine using threads, other parallel methods can be applied to GPUs and special-purpose hardware architectures. Also massive parallelism can be achieved using more systems and even a super-computer but the message passing and synchronization gets more complicated. Some work has been put into creating packages to deal with this messy message passing like Tachyon [4]. This software uses MPI to deal with the parallelism and synchronization but is made for bigger systems than a single computer. We choose to implement our own parallel system for optimizing ray tracing on one machine.

3 Adaptive Sampling

We used adaptive super-sampling to find where problem pixels were for fixing aliasing and improving shadows. Our method employed the basics of adaptive sampling, where the pixel of interest has rays traced through its corners and the variance between them finds if they need to be sampled further. If the variance between the rays does not exceed a set limit, the color of the pixel is set to the average of the four corner rays. The variance limit we set was 1%, which covered all of the problem pixels we encountered. See figure 1 for a representation.

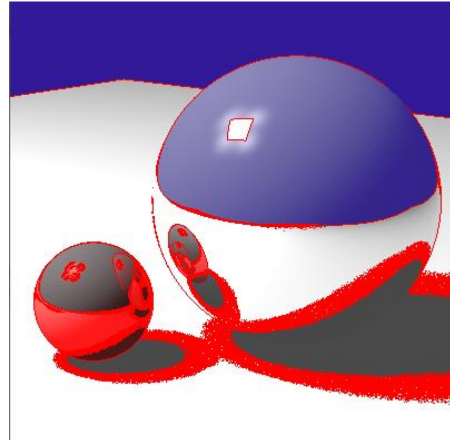

Figure 1: A screenshot of the problem pixels our algorithm finds. Problematic pixels are highlighted in red.

3.1 Adaptive Anti-Aliasing

Our initial attempts at adaptive anti-aliasing were to further sample the pixel by subdividing the pixel area into four equal quadrants. The variance between the four corners of the quadrant was then found and if the variance was too high it would be further sampled. It was limited to a depth of two for subdivisions to restrict the time spent on a single pixel since our focus was speed over quality. While this method granted a reduction of aliasing, we were able to create the same quality image by not performing the subdivision, but instead randomly sampling the pixel. This second method was faster, as we could get the same quality with 9 or less random samples as with calculating the heavily subdivided pixel.

3.2 Adaptive Shadow Sampling

The adaptive shadow sampling method we employed is nearly identical to the anti-aliasing method, but instead of sampling a pixel, we sampled the light source. The corners of the light source were tested for their variance, and any detected changes were treated as problems for the shading. If

all of the rays reached the light source then it was assumed that there were no obstructions. Likewise, if all of the rays

Figure 2: Erroneous soft shadows. Areas of the penumbra are divided into single shaded bands instead of smoothly blending.

didn't reach the light source it was assumed to be completely in shadow. Any variance of the rays casted at the light signaled that the shaded point was in the penumbra, where further calculations would take place. Treating the light source as a pixel in the same algorithm as the anti-aliasing created erroneous shadow effects, as in figure 2. We used the same solution as with anti-aliasing, where instead of performing sub-division on the area, we sent out additional rays at random points. When this occurred, the initial four ray casts were ignored for calculating the shading at that point. This solved the shadow rings from before, but there still remained artifacts of reflection around the reflection of the light source on the sphere. An enlarged screenshot of the light artifact is in figure 3. We were unable to find a solution for this artifact.

4 Parallelization

We utilized the fact that each ray being traced is independent of one another to easily parallelize the code. The use of multiple threads allows several rays to be computed at the same time limiting the negative effects of large numbers of samples for antialiasing and shadows. This

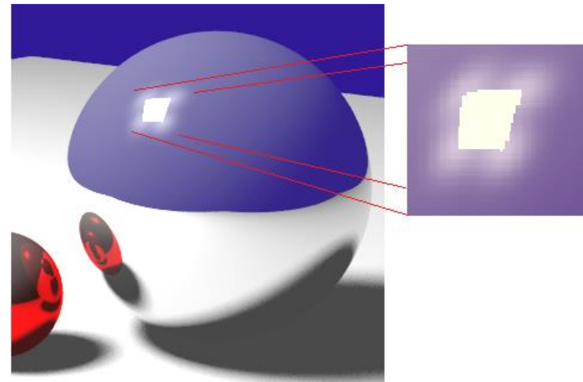

Figure 3: A closer look at the reflection of our light source. Notice the circular light reflections instead of a smooth bright spot.

parallelization was done on a single machine which limits the number of threads by the number of cores present on the machine. The implementation breaks into some number of threads and then the pixels are shared to a thread when it completes its computations. This reduces the bottleneck that occurs from specific pixels that take longer to render and allow other threads to run subsequent pixels. Threading was done outside of the ray creation such that any edits to the ray tracing algorithm would have no effect on the thread code. This design lets further optimizations be done without any knowledge of the parallel implementation. Our implementation also allows rendering to be done as computation is being completed due to the main thread being outside of the ray tracing.

5 Results and Discussion

We tested our renderer on a T410 Lenovo laptop with an i5 Intel dual-core and a NVS 3100m nVidia graphics card. Our results are listed in table 1. A sample size of '-' indicates that random sampling amounts were not used, and the adaptive sampling code was in place for that element. The naive version is our renderer with random sampling per pixel for antialiasing and

Build	Average render time (seconds)
Naïve with 9AA/9SS samples	262.859
Parallelized naïve with 9AA/9SS samples	121.559
Only anti-aliasing improvements with 9AA/9SS samples	192.231
Only soft shadows improvements with 9AA/9SS samples	88.983
Both anti-aliasing and soft shadows improvements with 9AA/9SS samples	99.001
Final implementation with 9AA/9SS samples	24.189
Final implementation with 16AA/16SS samples	34.375
Final implementation with 25AA/25SS samples	57.680

Table 1: A comparison of our average render times between builds. 9AA/9SS = 9 samples for anti-aliasing and 9 samples for soft shadow calculations.

random sampling for shadows. This is our base un-optimized renderer.

The improvements for adaptive anti-aliasing and soft shadows were similar in implementation, but the slight differences for the soft shadow calculations needed to deal with the light source caused them to have drastically different effects on the render time. The soft shadows improvements made a much more pronounced boost in efficiency compared to adaptive anti-aliasing.

Each of the improvements we made had a significant reduction in render time. This was usually a 2x speedup or better for a single optimization. Compared to the naive version, our complete implementation was 10x faster. The final rendering is shown in figure 4. While it is not real time, our methods still provide a massive increase to the speed for high quality images.

6 Summary and Future Work

We solved the problem of aliasing and implemented a soft shadow calculation in an efficient algorithm to make a fast ray tracer. For our sampling methods we found that high depth adaptive subsampling proved to provide not enough quality for the loss in efficiency compared to our no depth adaptive method. With the way we viewed our light sources, the method for anti-aliasing and soft shadows were nearly identical, although had different errors along the way. The inclusion of parallelization dramatically improved our render times thanks to the nature of ray tracing. Our method allows for any number of threads to be run, up to the limit of the number of processing cores in our hardware.

Further improvements to our project would include some sort of hit structure to speedup hit detection, and pushing our implementation to a more massive parallel capable machine. Either a hierarchical bounding volume structure or bounding grid could be used to avoid checking every object for each ray. To improve parallelization a movement towards clusters or supercomputers would involve the need for message passing and would raise the upper bound on threading.

References

[1] Jon Genetti and Dan Gordon.1993. Ray Tracing With Adaptive Super sampling in Object Space.Graphics Interface '93. pages 70-77.

[2] Whitted, T. 1980, An Improved Illumination Model for Shaded Display, Communications of the ACM 23(6): 343-349.

[3] Arjan J. F. Kok and Frederik W. Jansen. 1992. Adaptive Sampling of Area Light Sources
in Ray Tracing Including Diffuse Interreflection. EUROGRAPHICS '92 / A. Kilgour and L. Kjelldahl (Guest Editors), Blackwell Publishers

[4] Stone E. 1998, An Efficient Library for Parallel Ray Tracing and Animation. University of Missouri.

[5] Wald I. Purcell T. Schmittler J. Benthin C. Slusallek P. Realtime Ray Tracing and its use for Interactive Global Illumination. EUROGRAPHICS 2003 2-3

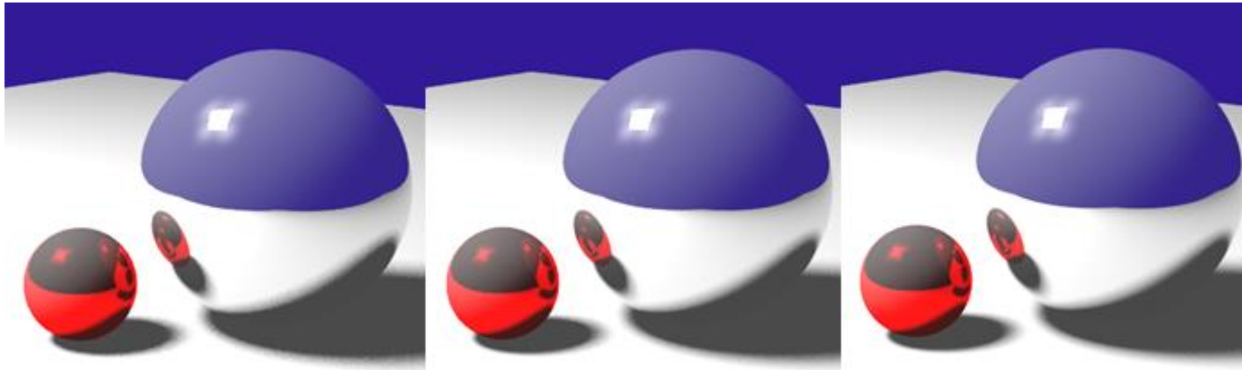

Figure 4: A comparison of our final renderings with different sample sizes. From left to right, 9 samples for anti-aliasing and 9 soft shadow samples, 16 samples for anti-aliasing and 16 soft shadow samples, and 25 samples for anti-aliasing and 25 soft shadow samples. At the highest quality only the soft shadows have a noticeable improvement compared to the lowest we ran.

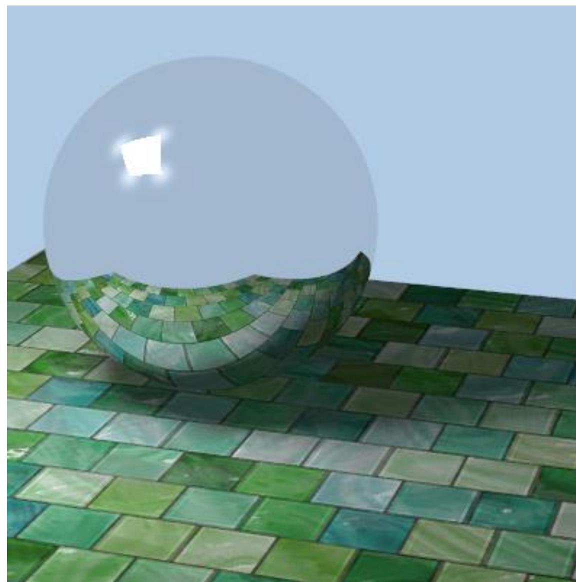

Figure 5: Screenshot of a different scene of a textured plane and a reflective sphere. 20 anti-alias and shadow samples were used. Average render time was 47.931 seconds