PERFORMANCE OPTIMIZATION OF PHOTON MAPPING FOR LIGHT ANALYSIS ON DIAMOND MODELS

Andrew Faulds

1 MOTIVATION

Our project is built on top of the work done by R. Nordhauser in "A Comparison of the Quality of Light Distribution in Different Diamond Models. This paper (her Master's degree thesis) attempts to analyze different models of diamond by comparing the amount of light sent out of the top of each diamond. Rebecca uses mesh geometry and photon mapping to trace photons throughout a scene with a diamond in it as they reflect and refract off of faces of the diamond. She collects statistics about the photons that escape through faces of the diamond to determine which model shoots the most light through the top.

There were two major bottlenecks in the code that were getting in the way of the analysis. The first was the tracing of photons throughout the scene. Tracing enough photons to gather accurate statistics for the project can take on the order of minutes or even hours. This impedes progress, not only because final results take a long time to calculate but debugging is slowed down by the large wait time. The second major bottleneck is the ray tracing of the image, which is important to obtain images of the diamond in the scene. The number of pixels that must have colors calculated and the photons that are gathered as part of the scene's indirect lighting both have a big impact on the rendering time, which also ranges between a few minutes and several hours. Our prediction was that improving the performance of both bottlenecks would help achieve results significantly faster.

2 RELATED WORK

Some efforts have been put into parallelizing ray tracing, but most involve doing accelerated ray tracing using a GPU [2][3]. Because this option was out of the scope of our project due to time and complexity constraints, we opted to parallelize ray tracing on the CPU. The project was already doing CPU-based ray tracing. Henrik Wann Jensen's work on photon mapping is the source of much of the material for this project [3]. His photon mapping technique and algorithm are used throughout the project. He also uses a KD-Tree balancing algorithm in [1], with which he improves rendering times by 50%. He references [4] as the paper from which he got the balancing algorithm. It also contains more information on optimizing KD-Trees.

3 METHOD

3.1 Code Profiling

The first optimization strategy used was code profiling. We analyzed the existing gather algorithm in the project to determine what improvements could be made to speed it up. The first problem was a lot of needless work being done gathering photons. The code would start off with a radius guess of 0.001, and increase itself by a factor of two each time if insufficient photons were gathered. The problem with this strategy is if the initial guess is quite far off, each ray will have to double the guess several times before even coming close to the radius that is required for sufficient photon collection. As shown in Figure 1, each ray that is traced ends up doing 5 or 6 times the work that is actually required to be done, significantly slowing down the rendering process. We made a change to the code to store the exact radius of the outermost photon from the current gather iteration. Each ray's initial radius "guess" is then set to slightly more than the last successful radius value that was stored (1.15 * *last_radius*). Since neighboring positions often have very similar radius values, this strategy saves a lot of computing time. There may be an occasional iteration that collects too few or too many photons, but the caching of the last radius that works ensures that future iterations will not also need to do the extra work. This strategy would be less effective on a scene with more complex geometry, since neighboring rays might not belong to the same surface and have very different radius values for gathering the correct number of photons.

```
Collecting Photons for point: -1.74373 7.74416 5

Round 0.001 and 0 photons found.
Round 0.002 and 92 photons found.
Round 0.004 and 92 photons found.
Round 0.008 and 92 photons found.
Round 0.016 and 92 photons found.
Round 0.032 and 92 photons found.
Round 0.064 and 92 photons found.
Round 0.128 and 92 photons found.
Round 0.256 and 92 photons found.
Round 0.512 and 92 photons found.
Round 1.024 and 162 photons found.
Collecting Photons for point: -2.22571 -2 3.74614

Round 0.001 and 0 photons found.
Round 0.002 and 82 photons found.
Round 0.004 and 82 photons found.
Round 0.008 and 82 photons found.
Round 0.016 and 82 photons found.
Round 0.032 and 138 photons found.
Round 0.064 and 138 photons found.
Round 0.128 and 138 photons found.
Round 0.256 and 138 photons found.
Round 0.512 and 138 photons found.
Collecting Photons for point: -1.72994 7.83432 5

Round 0.001 and 0 photons found
```

Figure 1: Debugging output from gather algorithm

The second problem we discovered using the code profiler was that a lot of time was being spent in the sort function. As part of the gather algorithm, the photons returned from the KD-Tree are sorted based on their distance from the ray's position in order to find the *n* closest photons to the ray's position. After using the code profiler to take a look at what was going on, it seemed like there was unnecessary time being spent in the comparison function. A closer look revealed that the sort's comparison function was calculating distance to the pixel position of both input Photons, and then returning true if the first was closer than the second. The distance was being re-calculated each time the comparison function was called,

wasting computation time. To speed it up, we pre-calculated the distance between each Photon and the radius point. We then changed the comparison function the compare the pre-calculated distances rather than re-compute them.

There were a few other minor code tweaks that we made after running the profiler. Before places in the code with many push_back operations, we manually resized vectors to avoid multiple automatic resize operations. We changed some pieces of code to use squaring instead of square-rooting, since unnecessary square root operations were slowing the sort down. The existing algorithm also erased values from vectors, which is quite inefficient, so we refactored it to work without erasing.

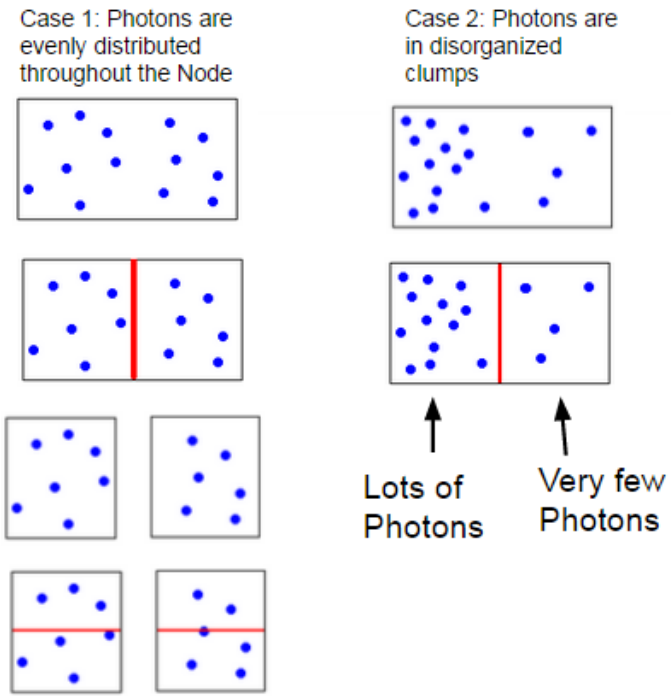

Figure 2: Result of splitting 2 different possible KD-Tree nodes using naïve splitting strategy

3.2 KD Tree Balancing

The second optimization we looked at was improving the performance of the photon gathering algorithm. A KD-Tree was used to store the photons that had been shot into the scene in order to efficiently perform the k-nearest-neighbor problem that is part of the algorithm. The KD-Tree contained photons that had positions inside of a bounding box representing the volume held within the KD-Tree node. When a node got too full, it would split exactly down whichever of its x, y or z axes was the longest. When the photons are fairly evenly distributed, this strategy works well, as it creates child nodes with about the same number of photons stored in each one. When the photons are clumped, as can be the case when the photons are the result of reflecting or refracting through a certain spot in the diamond model, this strategy leaves child KD-Tree nodes with unbalanced numbers of photons in them. This difference is displayed in Figure 2.

We used was a modified version of the KD-Tree balancing algorithm employed by Jensen in [1]. The algorithm works by selecting the median element of those to be inserted and then inserting those less than the median to the left, and those greater to the right. The result is two children of the current KD-Tree node with exactly the same number of photons in each.

The way that we implemented this algorithm requires that all of the photons be stored in a vector before KD-Tree construction (that is, they cannot be inserted on-the-fly). Because the project's existing algorithm inserted each photon on-the-fly into the KD Tree as each one was being traced throughout the scene, we had to modify the code to add the photons to a temporary vector instead. Once all photons had been traced, we used a recursive KD-Tree constructor to create a balanced KD-Tree from the vector containing all photons. The basic algorithm is as follows. Each node being constructed received the depth and vector of photons as arguments. The node would first determine if it needed to split, and if so, which axis was the longest. The node would then sort the photons by their position's coordinate along the longest axis. Finally, it would call the constructor for its two children; the first child being passed the first half of the parent vector (those points less than the median), and the second child being passed the second half of the parent vector (those points greater than the median).

The end result of using this balancing algorithm is that each KD Tree node has the same number of photons as each other node, eliminating inefficiencies resulting from querying nodes that have far too many photons in them.

3.3 Photon Tracing Parallelization

The third optimization that we implemented was parallelization of the photon tracing throughout the scene. Each photon is traced independently through the scene and intersected with the mesh geometry. Each intersection may result in a refraction or reflection, which causes another photon to be traced at an angle from the intersection. Photons are traced sequentially, and each photon has no effect on the tracing of other photons. As a result, the tracing can be easily parallelized. We used the C++11 STL thread library to use multiple threads, each of them tracing photons concurrently. The total number of photons is divided up evenly by the number of threads being used to do the tracing. The master thread spawns all the other threads and then waits for them to finish.

We had to add some mutex locks in order to prevent data race situations between threads. The global random number generator is not thread safe and needed locking around the rand() function. The other two data structures

that are shared between threads, the master vector of photons and ray tree visualization structure, also needed locking around calls to insert into them to ensure that only one photon was inserted at a time. Mutex locking can introduce significant additional overhead if threads spend a lot of time waiting for locks to become available, but running the code profiler determined that the photon tracing mutex locks did not significantly impact the performance of the algorithm.

3.4 Ray tracing Parallelization

The fourth and final optimization that we implemented was parallelization of the ray tracing of images to render. Like the tracing of photons, each ray is traced sequentially and each ray can be calculated independently without affecting other rays. As with photon tracing, the ray tracing can be easily parallelized. We also used the C++11 threading library for the ray tracing parallelization. Since the rendering is done a few hundred pixels at a time in the idle() function, it's a little more complex than the photon mapping parallelization. The master thread first creates a set number of threads. Each thread traces one full row of pixels from the rendered image, storing its vertex positions and colors in a vector. The master thread then joins the created threads in order, making openGL drawing calls for each thread's vertices and colors after it has completed. Because the threads are created and joined in the same order, the order of the pixels is preserved and the image looks the same as if it was rendered by a single thread, one pixel at a time. No locking was needed since each thread has its own vector to store vertices and colors in, and the master thread is the only thread making calls to openGL. Significant work was required to modify the rendering code to allow this structure; the previous structure involved a lot of static variables that would have caused threads to share variables that they needed local copies of.

4 Results

All tests were run on a Lenovo IdeaPad Y580 with Intel Core i7 2.30 GHz processor and NVIDIA 660M graphics card with 8 GB of RAM. Tests were compiled with g++ and run on Ubuntu 14.04.

4.1 Code Profiling Optimizations

We ran several test cases on the unoptimized and optimized code to determine the difference in time spent both tracing photons and rendering images with ray tracing. As shown in Figure 3, there was almost no performance

difference in the tracing of photons. This is to be expected, since most of the optimizations were made to the ray tracing code. However, the optimized ray tracing code ran in 33% of the time that the unoptimized version took to run.

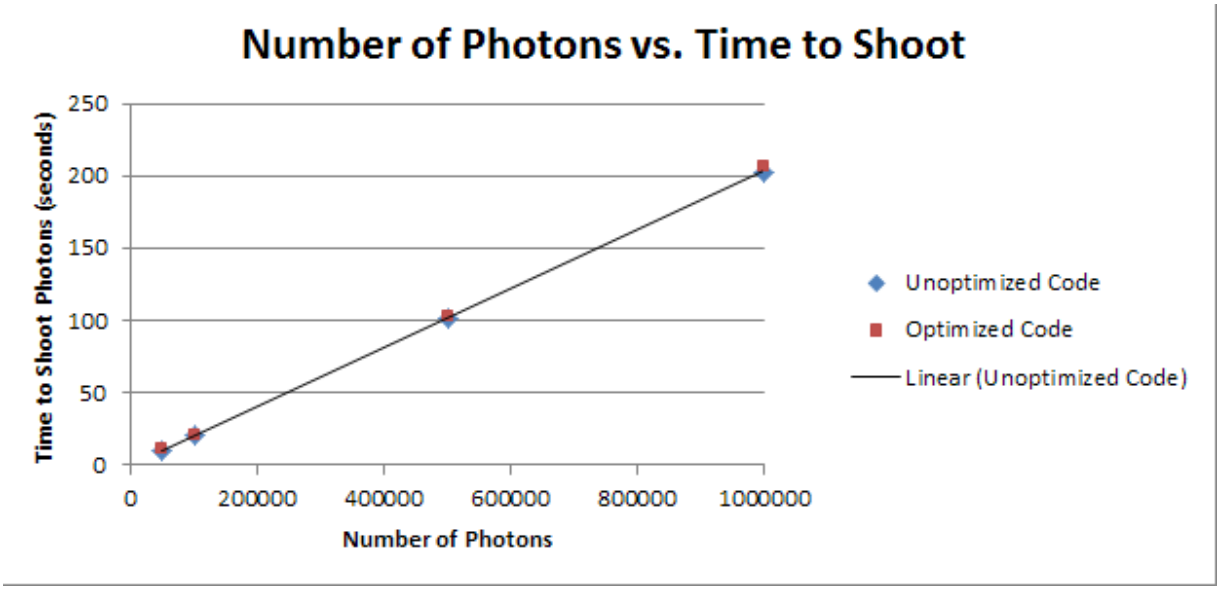

Figure 3: Number of photons traced vs. total time to trace for unoptimized and optimized code.

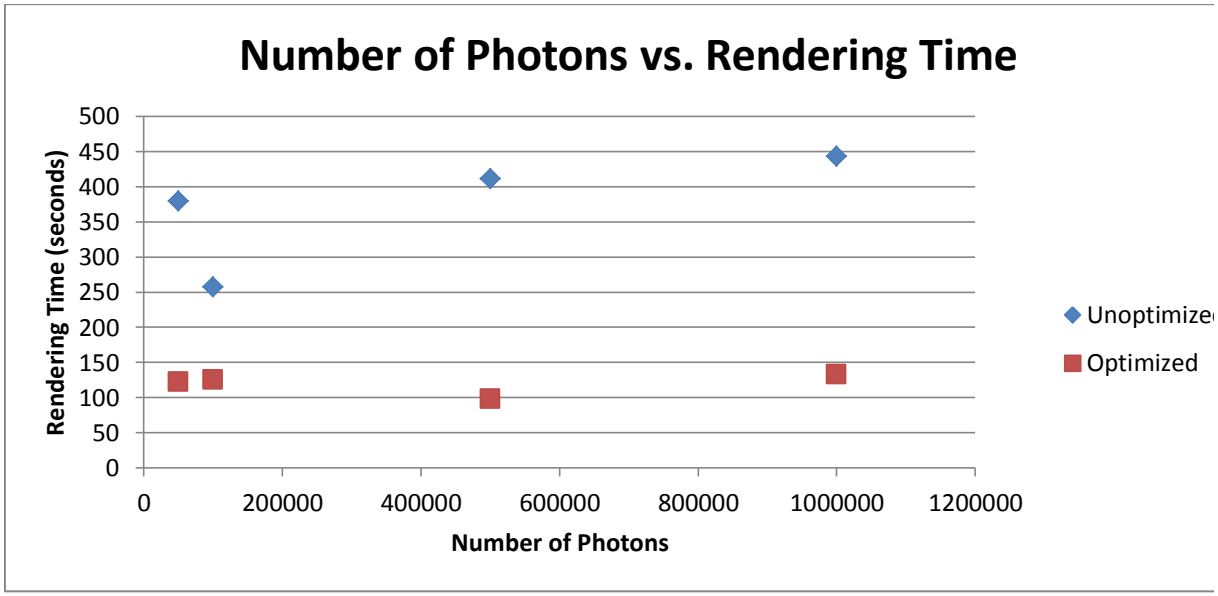

Figure 4: Number of photons traced vs. total time to render for unoptimized and optimized code.

4.2 KD-Tree Balancing

We present the results of running test cases on the versions of the code with the original, unbalanced KD-Tree and the KD-Tree after applying the balancing algorithm. The balanced KD-Tree took longer to construct than the unbalanced tree, as shown in Figure 5. This is to be expected, as a result of the step of the balancing algorithm that requires sorting the photons. The balanced KD-Tree resulted in a rendering time that was slightly higher than using the unbalanced tree. The balanced tree version does appear to scale better, as shown in Figure 6. If the trends displayed in those tests continued for higher numbers of photons, the balanced KD-Tree would be the better choice for best rendering performance.

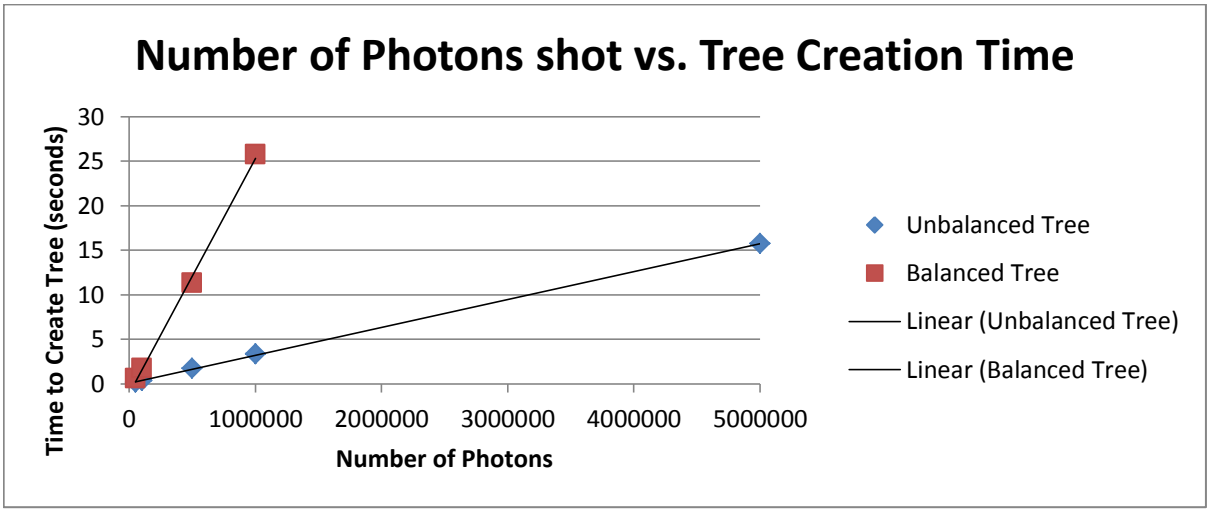

Figure 5: Number of photons traced vs. tree creation time for unbalanced and balanced KD-Trees.

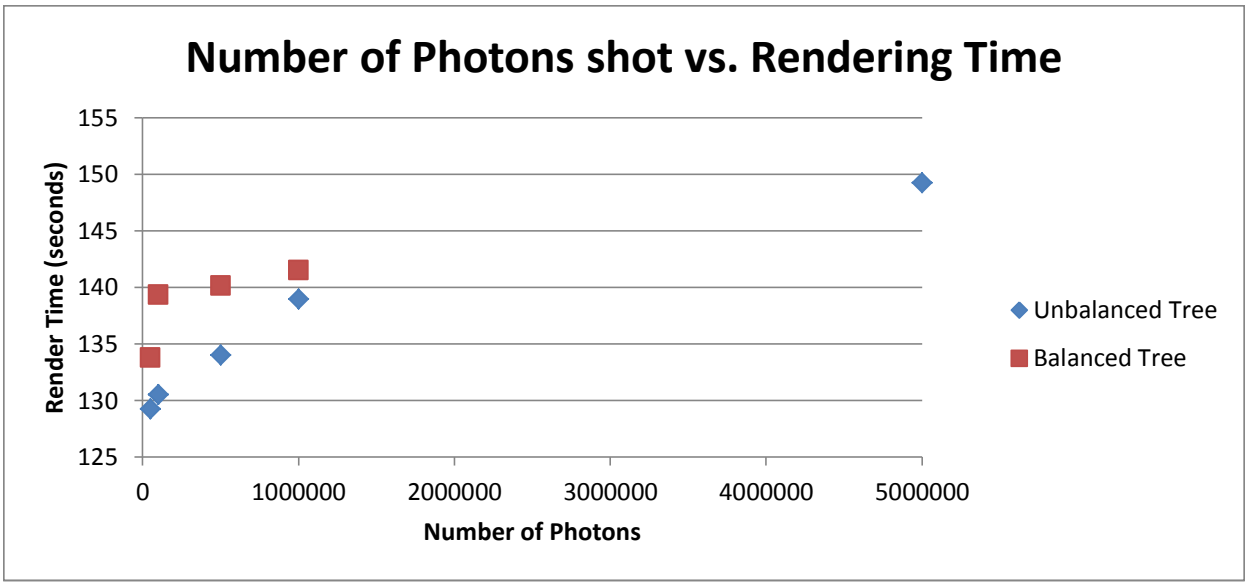

Figure 6: Number of photons traced vs. rendering time for unbalanced and balanced KD-Trees

4.3 Photon Tracing Multithreading

We ran several test cases on the multithreaded photon tracing code. We varied both the number of threads used and the number of photons traced throughout the scene. As shown in Figure 7, doubling the number of threads used does not quite cut the tracing time in half, as it would in the case of a perfectly parallelizable algorithm. Adding twice the number of threads tends run in 60% of the original run time, until a limit of 8 threads are reached, likely because the tests were run on hardware with 8 (4 hyper threaded) CPU cores.

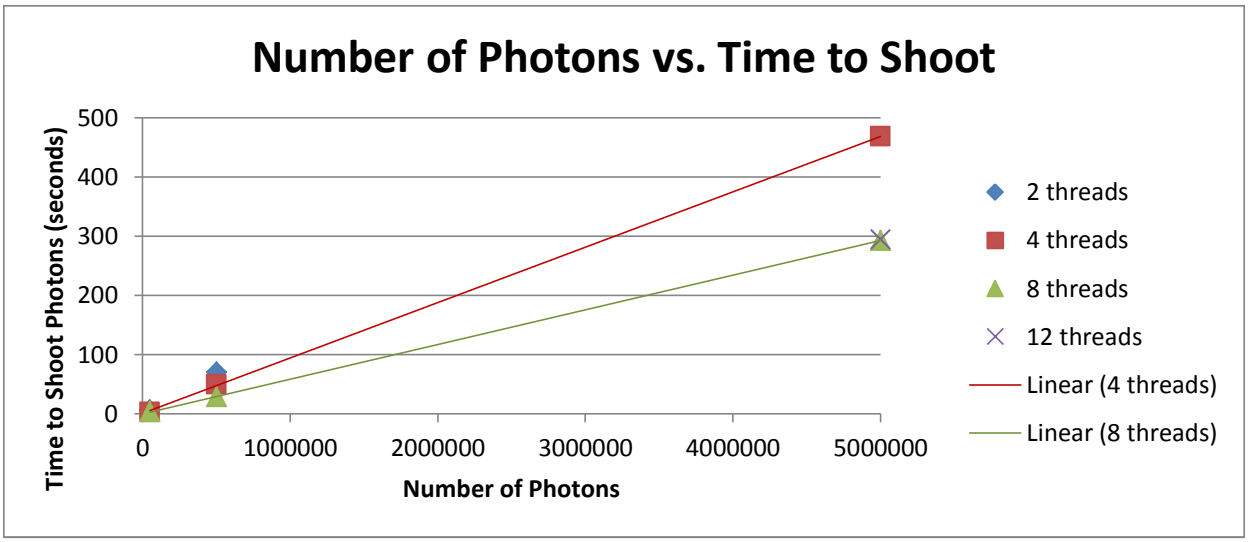

Figure 7: Number of photons traced vs. time to trace for varying numbers of threads

4.4 Ray Tracing Multithreading

We ran several test cases on the multithreaded ray tracing code as well. Like with the photon tracing tests, we varied both the number of threads used and the number of photons traced throughout the scene. The speed increase from 2 threads to 4 wasn't nearly as good as the increase from 4 threads to 8, as shown in Figure 8. Increasing the thread count from 2 to 4 improved the render time to 63% of the original. Using 8 threads resulted in a further improvement of 57% of the 4-thread runtime.

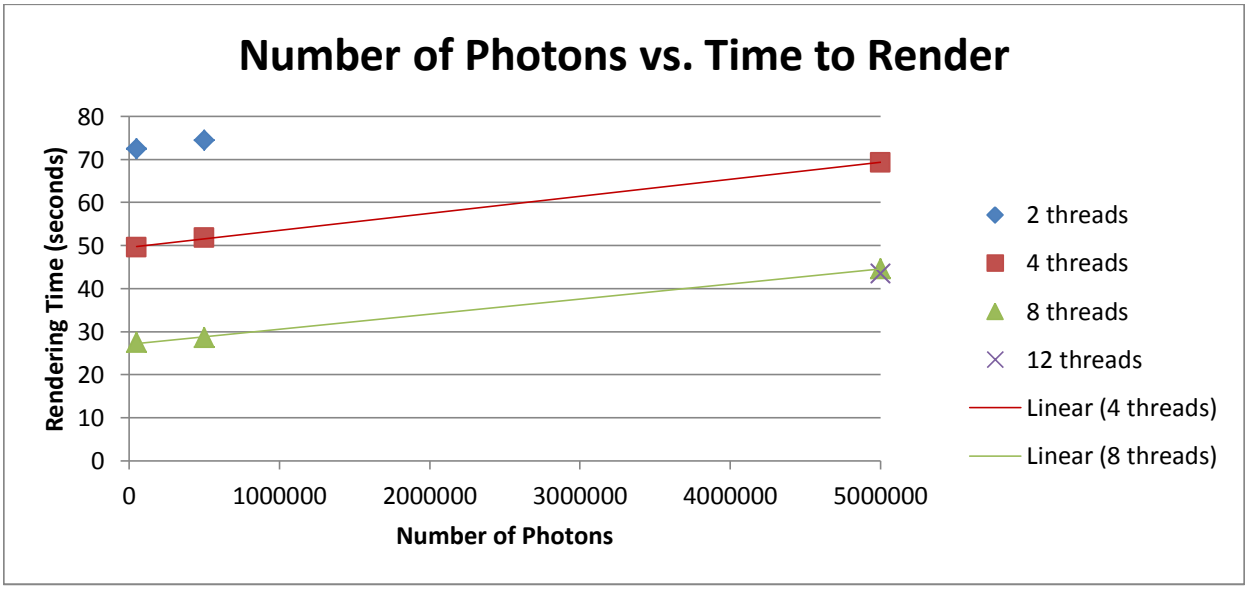

Figure 8: Number of photons traced vs. rendering time for varying numbers of threads

5 Conclusions

Several conclusions can be drawn from the work presented in this paper. First, small algorithmic changes can have a large impact on the performance of a project. Before any parallelism was done, the use of a code profiler and some small code tweaks cut rendering times in half. Carefully examining the algorithms being used and looking for more efficient ways to implement pieces of code is a much better strategy than blindly throwing more CPU cores at a problem. However, adding parallelism can also result in large performance increases once any algorithmic changes have been made.

6 Future Work

There are several possible directions for further work on this project. Parallelization could be improved by reducing the amount of mutex locking used. Mutex locks are used to save Photons when they're being traced, and to hold onto the global random number generator while it's generating numbers. Less locking overhead would mean less time spent waiting for locks, resulting in better parallel performance.

For more major performance increases, the project could be ported over to a GPU or to use MPI, and run on a supercomputer like RPI's Blue Gene/Q. The port to MPI would require significantly less effort, since its structure is quite functionally similar to the C++11 thread code from the project. In either case, larger test cases could be run in shorter amounts of time, though it's hard to predict the actual speedup without trying it.

The KD-Tree could be further optimized in a number of ways to speed up the gather indirect light operation of photon mapping. It could be stored efficiently as a binary heap as in [1], saving space needed for child pointers and eliminating pointer chasing. The KD-Tree has several constraints given to it – a maximum height, and a maximum number of photons stored by a Node before it can split. Tweaking these parameters may also result in better performance, though those avenues were not explored in this project due to time constraints. A faster sorting/median selection algorithm could also be implemented for the balancing algorithm.

7 References

[1] Henrik Wann Jensen: "Rendering Caustics on Non-Lambertian Surfaces". In Proceedings of Graphics Interface '96, pp. 116-121, Toronto 1996.

[2] de Greef M, Crezee J, van Eijk J C, Pool R and Bel A: Accelerated ray tracing for radiotherapy dose calculations on GPU Med. Phys. 36 pp. 4095–102, 2009.

[3] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. In SIGGRAPH ’02, pages 703–712, 2002.

[4] Bentley, Jon Louis: ”Multidimensional Binary Search Trees Used for Associative Searching”. Comm. of the ACM 18 (9), pp. 509-517, 1975.