# Optimized Ray Tracing for Single Scene Renders

Jenay Barela
Rensselaer Polytechnic Institute
Troy, NY
barelj@rpi.edu

Liam Roberts
Rensselaer Polytechnic Institute
Troy, NY
roberl6@rpi.edu

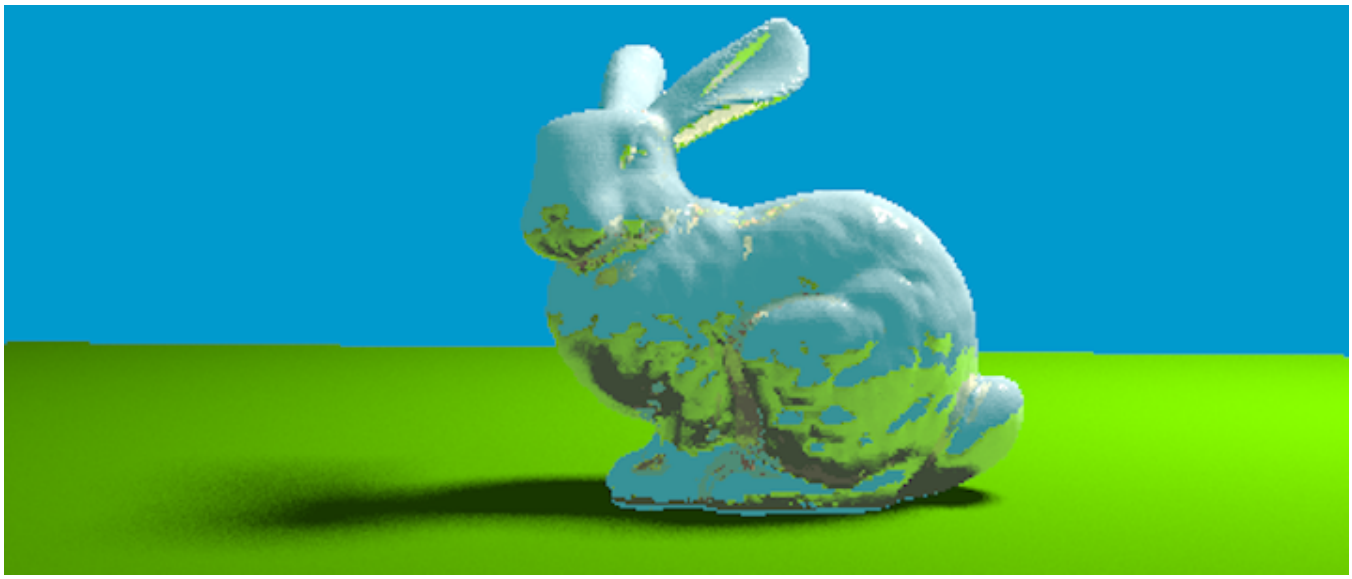

**Figure 1.** Bunny sitting in grass. Rendered on AiMOS

## Abstract

Ray tracing is a computationally expensive rendering technique used to create detailed images capturing scenes featuring complex geometry, lighting, and other effects. This method is typically constrained by its runtime, however this can be significantly reduced using parallel computing. We propose a novel, massively parallel, image rendering method used in the creation of ray-traced videos. Our method utilizes sample images to create a cache mapping color to on-screen positional data, reducing runtime when capturing images in-between various camera positions.

***CCS Concepts:*** • **Computing methodologies** → **Massively parallel algorithms**; **Ray tracing**; **Parallel programming languages**.

***Keywords:*** MPI, ranks, ray tracing, multi-threading, distributed computing, slurm

**ACM Reference Format:**
Jenay Barela and Liam Roberts. 2023. Optimized Ray Tracing for Single Scene Renders. In *Proceedings of Advanced Computer Graphics (CSCI 4530).* Rensselaer Polytechnic Institute, New York, NY, USA, 9 pages.

*CSCI 4530, May, 2023, Troy, NY*


## 1 Introduction

Ray tracing is a notoriously expensive rendering method. As such, much of the literature surrounding this technique focuses on simplifying or otherwise modifying effects such as soft shadows [7], refraction [24] and reflection [5, 20] such that they are optimized for runtime efficiency. Other approaches introduce parallel programming either through multi-threading [4] and/or distributing the workload across multiple processors [17]. Another technique involves classifying rays based on subdivided regions of space to accelerate object intersections [1]. However, these approaches typically optimize for some variant of classic ray tracing or implement completely alternate techniques for the purpose of capturing static images.

This paper addresses these major drawbacks of the above methods by using parallel programming, color caching, and ray tracing to render complex effects without suffering the traditionally coupled runtime costs. While these previous algorithms display significant advancements in the fields of computer graphics and parallel programming, some applications such as video production demand many ray traced frames to be produced in batch. These applications are the

source of our motivations for this project and are the recipients of the speedups our approach provides. However, a major drawback to our approach is the introduction of rendering artifacts. These artifacts can be reduced by decreasing the camera movement between frames and/or providing additional sample images to the algorithm.

## 2 Related Works

Of all of the components used in our approach, the most integral of them is ray tracing. Previous literature [8, 23] offer an in depth explanation of exactly what ray tracing is and the effects it can generate. Efforts to optimize this method have been a topic of research since ray tracings conception. Ray tracing in parallel and across multiple nodes has accounted for a large portion of this research, [13] however, these studies were limited due to the compute power the researchers had available to them. Other works in this area [2] have utilized message passing to create more complex parallel systems through information distribution. In our implementation, a similar type of caching is used to avoid unnecessary ray tracing. Work [11] has been done specifically to enhance the run time of passing data between processes similar to our application such as which looks into minimizing the unnecessary copying of messages between processes.

## 3 Ray Tracing

Ray tracing has a relatively simple implementation, described by the following pseudocode:

**Algorithm 1** Trace Ray

```
Cast ray into scene
for objects in scene do
    if Ray intersects object then
        Intersect with lights                ▷ Cast shadows
        for number of refection bounces do
            Trace Ray(x,y)                   ▷ Cast reflected ray
        end for                              ▷ Additional affects can go here
    end if
end for
```

Ray tracing involves firing one or more rays through the center of each pixel, $x_i$, $y_j$, from the camera (eye position) and returning color data regarding any hit objects. The color data collected from these rays is drawn to the screen. Additionally, users can layer additional effects such as shadows, anti-aliasing [6], refraction, reflections and participating media [16] to increase rendering quality and realism. Each additional stacked effect massively increases the run-time of the algorithm since certain effects such as reflection cause rays to be traced recursively and other effects such as anti-aliasing require multiple rays to be cast per pixel.

When rendering scenes involving complex geometry, runtime is slowed further by the base algorithms need to check each rays intersection with each primitive object. Spatial data structures such as KD-Trees [3] and Bounding Volume Hierarchies [12] have been used to speedup primitive ray collisions. However we chose to omit these data structures in our implementation in order to focus on the speedups provided by our parallelism and other accelerative methods.

### 3.1 Complex geometry

Despite not implementing a spatial data structure for ray intersections, we still require complex geometry to be displayed in our scenes. We choose to implement fast triangle intersections, avoiding the expensive step of calculating whether a ray intersects a plane [19]. It accomplishes this by mathematically translating one of the triangle's vertices to the origin and aligns the other two with the $y$ and $z$ plane. It then also translates the ray such that it aligns with the $x$ axis. This is visually demonstrated in Figure 2.

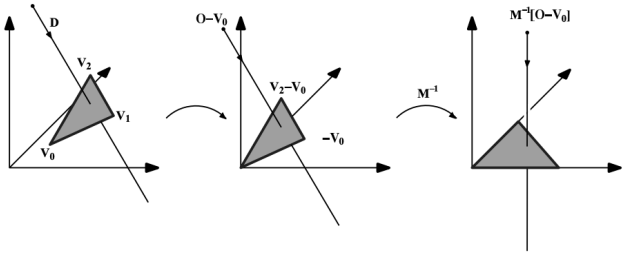


**Figure 2.** Transformation of triangle and ray [19]

## 4 Caching

As a camera pans across a scene, many of the pixels in intermediary frames share the same (or approximately the same) pixel color throughout the motion. However, conventional methods are forced to ray trace each of these images individually. To avoid this unnecessary work, our algorithm caches information from two fully ray traced images (those at the beginning and end of the camera's panning movement) to infer pixel color values of intermediary frames. As our algorithm ray traces the sample images, it intermittently stores pixel information into a global cache.

Our cache is a hash table mapping a pair of doubles (representing an (x, y) coordinate) to a vector of Pixels. A 'Pixel' is a struct containing an $(x, y)$ coordinate, the RGB value of its corresponding $(x, y)$ location on screen, and an id storing which frame 0...*num_samples* in the movement the data was cached from. Using this data, the intermediary frames compare the colors of cached pixels and the distance traveled by the ray before an intersection. At first, we attempted to make pixel color inferences by only using distance traveled before intersection.

However, this method had a tendency to leave artifacts on surfaces with gradient colors, such as the ball objects. To avoid these artifacts, we added a comparison of the cached

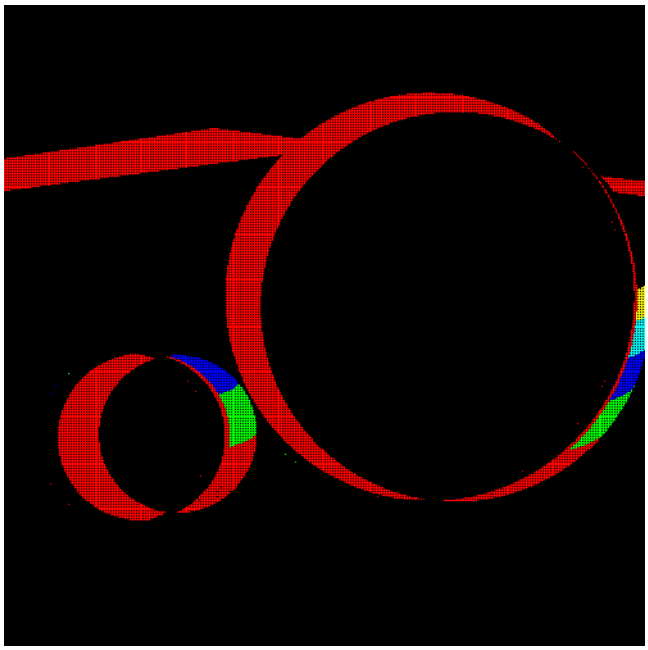

**Figure 3.** Color map when inferring pixel color using travel distance of rays. Color indicates sections where the distance travelled between sample image and frame is different. Red was the closest, followed by green, followed by blue.

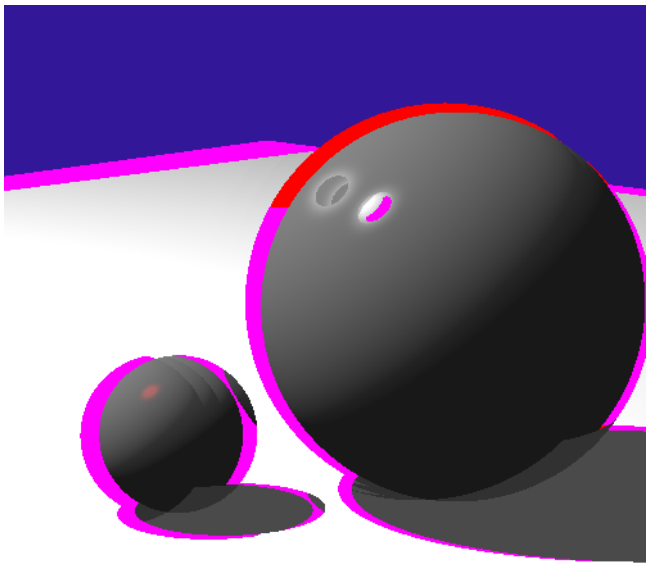

**Figure 4.** Example artifacts from too lenient of a color distance epsilon. The white circle on the top of the larger ball is a direct result of this. Highlighted colors indicate re-ray traced pixels

pixel colors. As we did this, we discovered that we needed a epsilon that allows for some minor change in color. If the epsilon was not used when comparing RGB pixel values in our cache, subtle changes in color created additional artifacts (4).

### 4.1 Artifact Reduction

Our approach allows for users to upload sample reference images to increase cache hits. This leads to reduction in both runtime and quantity and severity of artifacts as show in Figure 5.

## 5 Parallel Computing

Our approach utilizes MPI, Pthreads, and parallel I/O with a high degree of interaction to supply information to a globally distributed cache. This cache contains data relating to user provided sample images. Once the provided sample images have been uploaded, their data is distributed to all MPI ranks by the root process directly into respective child process cache. These ranks periodically send each other sections of their local cache in order to minimize duplicate work across other processes. Each MPI rank produces one or more frames of the final output video through a combination of cache look ups and ray tracing.

If a process cannot pre-determine the color of a location on screen via a cache lookup, it is ray traced and added to $\text{rank}_k$'s local cache, we refer to the results of these new ray traces as 'discoveries'. Once $\text{rank}_k$ has made enough discoveries, it distributes this newfound data to all other ranks, who then add it directly into their local cache for future use.

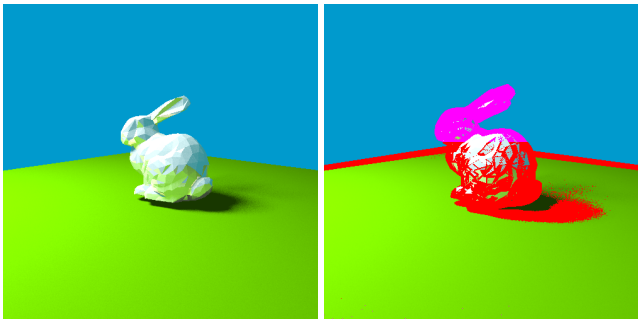

**Figure 6.** Still frame of rendered video (left) vs visualization of cache hits (right). Pink pixels indicate a re-ray traced location. Red pixels indicate cache hits. No highlighting indicates the resulting pixel color matched between the first and final frame

### 5.1 MPI

Each MPI rank is assigned to render a variable number of frames. Each of these ranks has a local Image object containing RGB data in a form that can be easily written to a file either serially or in parallel.

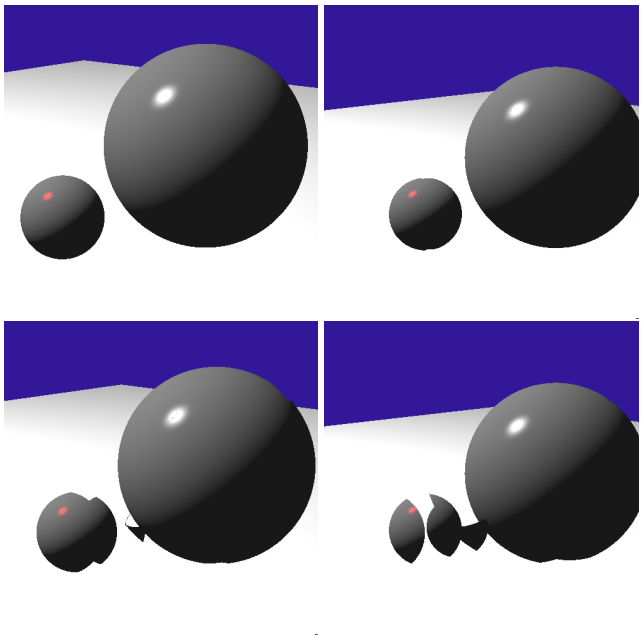

**Figure 5.** Example progressive artifact reduction via additional (pre-rendered) image uploads. In order from top left to bottom right: fully ray traced frame, frame with three uploaded reference images, one uploaded reference image, no reference frames

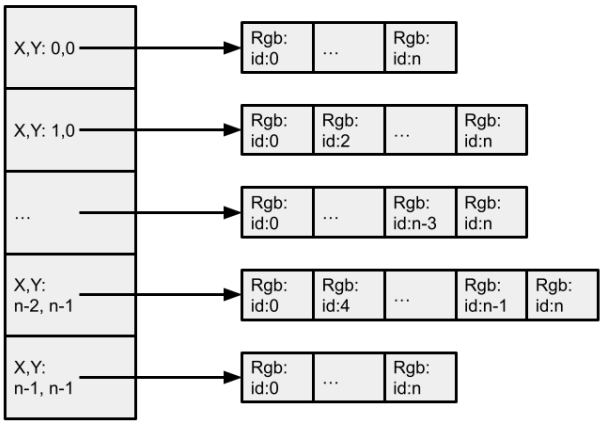


**Figure 7.** Internal representation of our cache

**5.1.1 Image Cache Lookup.** While rendering any frame, MPI $\text{rank}_k$ checks its local cache for matches. If a match is found, the color data is taken from this match, if not, a new ray is fired into the scene and the resulting color is cached and distributed to other processes.

**5.1.2 Cache Data Distribution.** Once $\text{rank}_k$ generates enough 'discoveries', these hash table entries are broken down into primitive types and distributed through an MPI broadcast call to all other processes. Each of these processes then reconstructs the hash table entries and inserts them into their local cache to be used for future use. Our current implementation of this data distribution is not optimal. The time demanded for inter-processor communication between two (or more) MPI ranks varies based on the size of the message being sent as well as the processes sending and receiving them [14]. As such, our application should allow for user-specified processor maps [14] to optimize inter-processor communication. Additionally, when an MPI rank reaches enough discoveries in our algorithm, it distributes all of them to all other ranks despite the fact that some of these ranks have no need for this cached data. Alternatively, we could implement a communication algorithm that allows ranks to pre-compute their 'ray traced' areas. These ranks would then lodge a global request to see if any other ranks have ray-traced that specific region. Algorithms utilizing proxy processes allowing for optimized communication between threads would be an ideal candidate for this purpose.[21]

**5.1.3 Checkpointing.** Our algorithm implements checkpointing in the form of periodic image saving once a variable number of images are finished rendering. When combined with parallel image uploading, this behaviour ensures that jobs can be paused and resumed with little to no additional run time. Other works have explored non-blocking checkpointing through the use of a scheduler and markers [9], however checkpointing after every batch of images is sufficient for our application as the cost of re-computing some lost data is less than the overhead of implementing a new checkpointing algorithm.

## 5.2 Multi-Threading

We implement multi-threading for ray tracing as a second form of parallel computing. Threads are created and dispatched to each pixel $x_i$, $y_j$ when rendering an image. The color data collected by these threads is aggregated, sent to the local cache, and used to set the color of the pixel within the local image object.

Before choosing Pthreads for our implementation, we first attempted to use CUDA. We quickly ran into problems as we attempted to re-structure our program (originally designed to ray trace one image at a time completely serially) to allow for the use of CUDA. After multiple attempts to rewrite the code base, we determined that we would have to restructure the entire application to implement it. Our application creates batches of threads, dispatches them across a portion of the image and then joins them back into the calling process so their discoveries can be shared with other processes.

In testing, we observed that the more weakly scaling effects we applied to each rendered frame, the greater our relative speedup when using Pthreads. However, in images that had little to no visual effects, the non-threaded program typically ran faster. This is likely due to the overhead of creating and managing threads outweighing the cost of

simply rendering these frames in serial. Due to this, we ran into problems running multi-node jobs with Pthreads on the AiMOS supercomputer as they would timeout as a result of the job time limit.

## 6 Results

We have run our benchmark tests on AiMOS across multiple nodes and ranks on the el8-rpi cluster. To account for the thirty minute time limit for submitted jobs on the supercomputer, we ran many of our larger tests with the reflective_spheres.obj file, the same test as displayed in Figure 5. However, other tests were conducted on scenes with significantly more complicated geometry.

**Figure 8.** Example of complex geometry used in our serial benchmark test

Our approach when rendering multiple frames of scenes with complex geometry significantly outperforms ray tracing each frame of the same scene individually. In Figure 9, a visually demanding test case, the serial implementation took 70 minutes to generate 16 frames while our approach computed the same 16 frames in 13 minutes.

### 6.1 Strong Scaling

In our strong scaling benchmarks, we computed 256 intermediary frames across different ranks using reflective_spheres.obj as our test input.

**Table 1.** Strong Scaling Across Different Problem Sizes (Time in Seconds)

| # Ranks | 32 Frames | 64 Frames | 128 Frames |
|---|---|---|---|
| 4 | 56.7 | 104.8 | 341.7 |
| 8 | 38.2 | 66.9 | 130.3 |
| 16 | 30.9 | 49.8 | 84.0 |
| 32 | 28.9 | 42.9 | 72.7 |

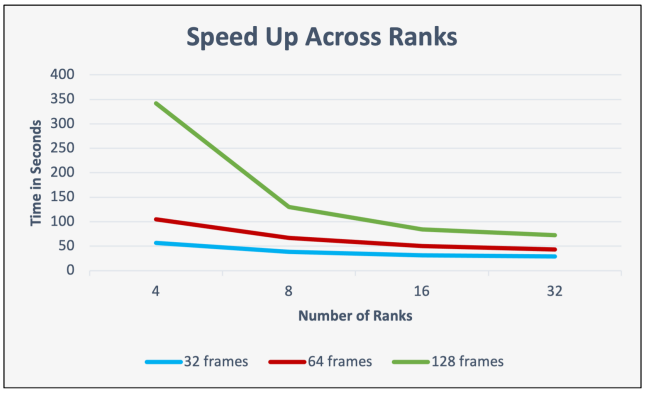


**Figure 9.** Plot of Table 1

Table 1 demonstrates our applications ability to strongly scale with a given problem set. While our multi-node jobs timed out on larger cases, our single node results show strong scaling has been achieved as we increase the number of MPI ranks used to compute a variable number of frames of the same test scene.

### 6.2 Weak Scaling

For the below weak scaling test, we increased the number of samples taken with the number of ranks remaining constant on the same test case.

**Table 2.** Weak Scaling

| # Ranks | Frames | Time |
|---|---|---|
| 16 | 16 | 13.3 |
| 32 | 32 | 29.8 |
| 64 | 64 | TIMEOUT |
| 128 | 128 | TIMEOUT |
| 256 | 256 | TIMEOUT |

Table 2 displays our results from our weak scaling test case across multiple nodes. Our test cases across multiple nodes timed out before any output was produced. This is likely due to a massive increase of traffic on AiMOS over the week prior to the projects deadline. In its current state, our algorithm sends data from each process to every other process. This may be the source of our multi-node slowdowns as all single node cases scaled as expected.

Using the same test case as above, but holding the number of ranks to one, we can clearly see how our cache improves our running time.

**Table 3.** Weak Scaling (single node)

| # Ranks | Frames | Time |
|---|---|---|
| 1 | 1 | 14.2 |
| 1 | 2 | 17.7 |
| 1 | 4 | 25.3 |
| 1 | 8 | 39.9 |
| 1 | 16 | 70.9 |
| 1 | 32 | 129.9 |
| 1 | 64 | 242.7 |
| 1 | 128 | 479.33 |
| 1 | 256 | 1065.98 |

Table 3 displays that one node rendering multiple frames does not create a linear time increase as frames are added to the job. This is due to the rendering processors accesses to previously cached data from prior frames. The utilization of this data means it is able to re-use work and thus spend less time ray tracing.

**Table 4.** Weak Scaling (single node cont.)

| # Ranks | Frames | Time |
|---|---|---|
| 32 | 32 | 30.9 |
| 32 | 64 | 41.6 |
| 32 | 128 | 69.2 |
| 32 | 256 | 146.5 |

Table 4 displays a larger weak scaling case in which 32 MPI ranks are assigned to an increasing number of frames. This data displays a roughly linear increase in time spent rendering as we increase the number of frames of a job.

The below scaling experiment shows how increasing image effects scales with sixteen ranks and the same number of image samples.

**Table 5.** Weak Scaling (VFX)

| # Shadows | Bounces | Time |
|---|---|---|
| 0 | 0 | 265.9 |
| 10 | 1 | 279.82 |
| 25 | 2 | 296.1 |
| 50 | 4 | 321.44 |
| 100 | 8 | 368.75 |

In Table 5 we can see how the time for the full render starts to rise as the number of effects increases. It appears that we have achieved a logarithmic function when plotting time to number of applied visual effects. This intuitively makes sense since most visual effects are clustered in specific parts of an image/scene. Thus, as more visual effects are added, they do not increase the quantity of pixels that need to be re-ray traced.

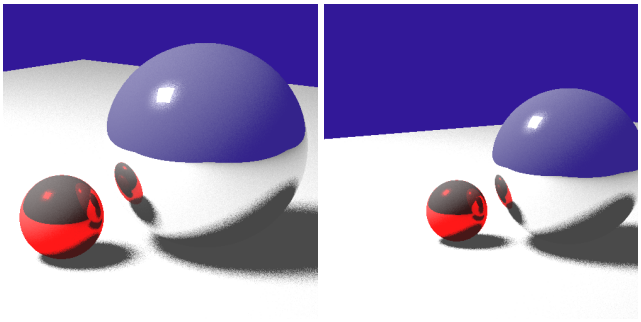

**Figure 10.** Example start and ending images of the 'Weak Scaling (vfx)' tests with two bounces and ten shadow samples

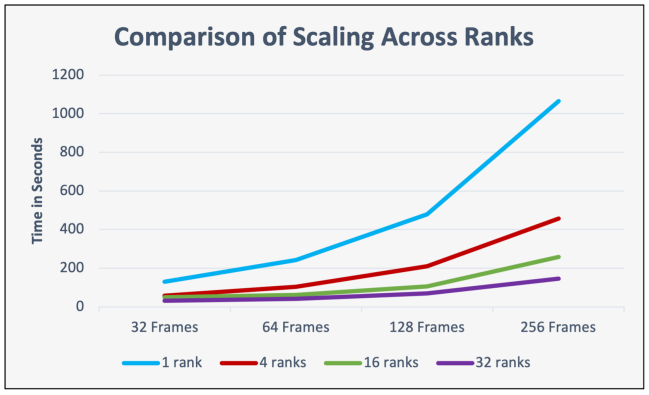


**Figure 11.** Graph of the results from scaling the number of images compared to different numbers of ranks

## 7 Conclusion

We have presented a novel algorithm for rendering ray-traced image samples used in creating high frame rate videos, optimized through massively parallel multi-processor communication. The method is based on previous works relating to inter-processor caching [18] and multiprocessor information optimization [22]. By allowing users to upload sample ray-traced images, we improve rendering time for multi-frame batched jobs. The resulting method provides a speedup when compared to conventional methods and our test-images demonstrate that little information is lost in the inference process utilizing the globally updated cache. This method is therefore useful in applications requiring large quantities of ray-traced images of the same scene to be rendered.

### 7.1 Limitations

**7.1.1 Run time.** Even when fully optimized, ray tracing is an expensive algorithm when multiple effects are used in tandem. Due to the time limits set on our jobs by AiMOS, we were largely unable to time these large-scale tests on multiple nodes. This left us with relatively primitive benchmarking scenes that work for the purposes of testing our relative speedup, but leave us with lacking visuals and test variety.

**7.1.2 Artifacts.** Currently, this technique causes severe artifacts in certain test cases, specifically cases in which multiple objects are clustered in space. Ideally, the algorithm should intuitively tell which regions of the image contain clustered objects and manually ray-trace these portions, perhaps even allowing for users to specify these regions ahead of time. This would involve supplying each rank with scene data as well as re-working the existing ray tracing architecture to accommodate this spatial component.

**7.1.3 Lack of Resources.** As we preformed experiments, we ran in to many problems caused by the lack of resources available on AiMOS. Not only did it take hours, and occasionally days to run jobs that required more than one node, but often these jobs would either error out due to system problems out of our control or exceed the short time limit. When running tests locally to gauge expected run times across multiple nodes, we would get varying run times from twenty seconds to over twenty five minutes for the same test case. When running these same test cases through a slurm job, they would almost always time out. We did our best to try and find the source of these issues, but we were unable to find the source of these problems in our code. Without a time limit, our code would be able to successfully complete its render on all of these larger-scale test cases.

### 7.2 Future Works

**7.2.1 Parallel Improvements.** Ideally, the work done by ray tracing should be done in parallel by CUDA threads. This would massively speedup the program and allow for greater testing suites capable of scaling both ranks and threads. Currently, our application optimizes by using cached color data of provided images. Our main algorithm could be expanded to make more informed decisions based off of ray intersection data, such as what type of material a ray hit and how far the ray traveled. Another area of improvement would be to implement a more advanced method of inter-processor communication such as nearest-neighbor communication [15] to prevent excess information sharing between nodes.

**7.2.2 Graphics Implementations.** While we currently use fast triangle intersections [19], the use of an spatial data structure such as a Bounding Box Hierarchy [3] or Bounding Volume Hierarchy [12] could be implemented to speedup scenes containing complex geometry.

Currently, all frames are captured on a straight line from the starting position to the ending position with the same focal point. Our application allows for users to move the camera along a B-spline curve [10] computed using four points provided by the user. This implementation is not yet finalized and results in unexpected path behavior. As such, it is not included in the work of this paper. However, this can be improved on in the future to offer more engaging and varied camera paths. Additionally, a working and accurate refraction implementation would provide additional effects to test.

### 7.3 Distribution of Work

**7.3.1 Jenay.** Responsible for saving .ppm images, parsing of triangle .obj files, and fast triangle intersections. Created initial cache pre-parallelization as well as cache comparison algorithm. Implemented multi-threaded git branch of code via Pthreads, and B-spline algorithm.

**7.3.2 Liam.** Responsible for MPI parallelization, cache representation in parallel/multiprocessor communication on AiMOS, refraction, sample image uploading, parallel image upload, CUDA discovery and re-write phase, data synchronization across processors.

**7.3.3 Both.** Paper Figures (graphics and timing data), CMake project building on AiMOS, artifact reductions, writing of paper, relevant literature readings, checkpointing, re-working of pre-existing code to allow for parallel implementation, github version control.

## References


[1] James Arvo and David Kirk. 1987. Fast ray tracing by ray classification. *ACM Siggraph Computer Graphics* 21, 4 (1987), 55–64.

[2] Didier Badouel, Kadi Bouatouch, and Thierry Priol. 1994. Distributing data and control for ray tracing in parallel. *IEEE computer graphics and applications* 14, 4 (1994), 69–77.

[3] Jon Louis Bentley. 1975. Multidimensional binary search trees used for associative searching. *Commun. ACM* 18, 9 (1975), 509–517.

[4] Brian C Budge, John C Anderson, Christoph Garth, and Kenneth I Joy. 2008. A straightforward CUDA implementation for interactive ray-tracing. (2008), 178–178.

[5] Nathan A Carr, Jared Hoberock, Keenan Crane, and John C Hart. 2006. Fast GPU ray tracing of dynamic meshes using geometry images. 2006 (2006), 203–209.

[6] Edwin Catmull. 1978. A hidden-surface algorithm with anti-aliasing. *ACM SIGGRAPH Computer Graphics* 12, 3 (1978), 6–11.

[7] Eric Chan and Frédo Durand. 2003. Rendering Fake Soft Shadows with Smoothies. (2003), 208–218.

[8] Robert L Cook, Thomas Porter, and Loren Carpenter. 1984. Distributed ray tracing. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*. 137–145.

[9] Camille Coti, Thomas Herault, Pierre Lemarinier, Laurence Pilard, Ala Rezmerita, Eric Rodriguez, and Franck Cappello. 2006. Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant MPI. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. 127–es.

[10] Carl De Boor. 1972. On calculating with B-splines. *Journal of Approximation theory* 6, 1 (1972), 50–62.

[11] Andrew Friedley, Greg Bronevetsky, Torsten Hoefler, and Andrew Lumsdaine. 2013. Hybrid MPI: efficient message passing for multi-core systems. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 1–11.

[12] Stefan Gottschalk, Ming C Lin, and Dinesh Manocha. 1996. OBBTree: A hierarchical structure for rapid interference detection. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*. 171–180.

[13] S.A. Green and D.J. Paddon. 1989. Exploiting coherence for multiprocessor ray tracing. *IEEE Computer Graphics and Applications* 9, 6 (1989), 12–26. https://doi.org/10.1109/38.41466

[14] Torsten Hoefler, James Dinan, Darius Buntinas, Pavan Balaji, Brian Barrett, Ron Brightwell, William Gropp, Vivek Kale, and Rajeev Thakur. 2013. MPI+ MPI: a new hybrid approach to parallel programming with MPI plus shared memory. *Computing* 95 (2013), 1121–1136.

[15] Torsten Hoefler and Jesper Larsson Traff. 2009. Sparse collective operations for MPI. In *2009 IEEE International Symposium on Parallel & Distributed Processing*. IEEE, 1–8.

[16] Henrik Wann Jensen and Per H Christensen. 1998. Efficient simulation of light transport in scenes with participating media using photon maps. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*. 311–320.

[17] Sadraddin A Kadir and Tazrian Khan. 2008. Parallel ray tracing using mpi and openmp. *Project Report, Introduction to High Performance Computing, Royal Institute of Technology, Stockholm, Sweden* (2008).

[18] Tom Loos and Randall Bramley. 1996. MPI performance on the SGI Power Challenge. In *Proceedings. Second MPI Developer's Conference*. IEEE, 203–206.

[19] Tomas Möller and Ben Trumbore. 2005. Fast, minimum storage ray/triangle intersection. In *ACM SIGGRAPH 2005 Courses*. 7–es.

[20] Frederic H Pighin, Dani Lischinski, and David Salesin. 1997. Progressive Previewing of Ray-Traced Images Using Image Plane Disconinuity Meshing. *Rendering Techniques* 97 (1997), 115–125.

[21] Srinivas Sridharan, James Dinan, and Dhiraj D Kalamkar. 2014. Enabling efficient multithreaded MPI communication through a library-based implementation of MPI endpoints. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 487–498.

[22] Daniel M Wadsworth and Zizhong Chen. 2008. Performance of MPI broadcast algorithms. In *2008 IEEE International Symposium on Parallel and Distributed Processing*. IEEE, 1–7.

[23] Turner Whitted. 2005. An improved illumination model for shaded display. In *ACM Siggraph 2005 Courses*. 4–es.

[24] Chris Wyman. 2005. An approximate image-space approach for interactive refraction. *ACM transactions on graphics (TOG)* 24, 3 (2005), 1050–1053.

# 8 Blooper Images

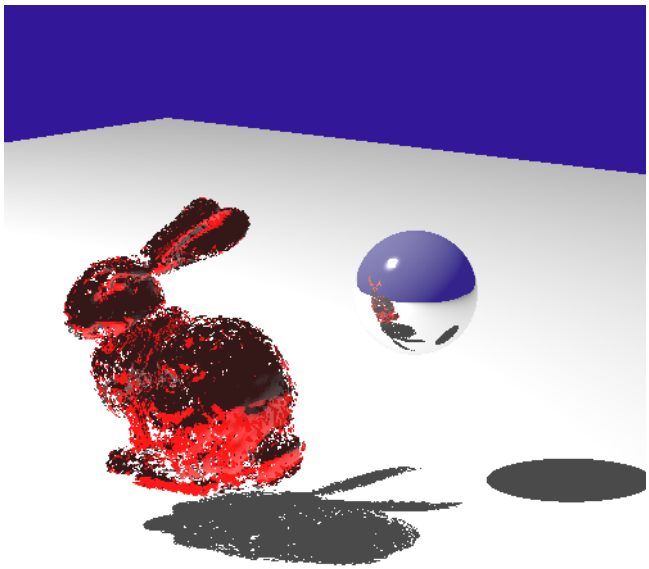

**Figure 12.** Holey bunny due to too big of an epsilon during triangle intersection

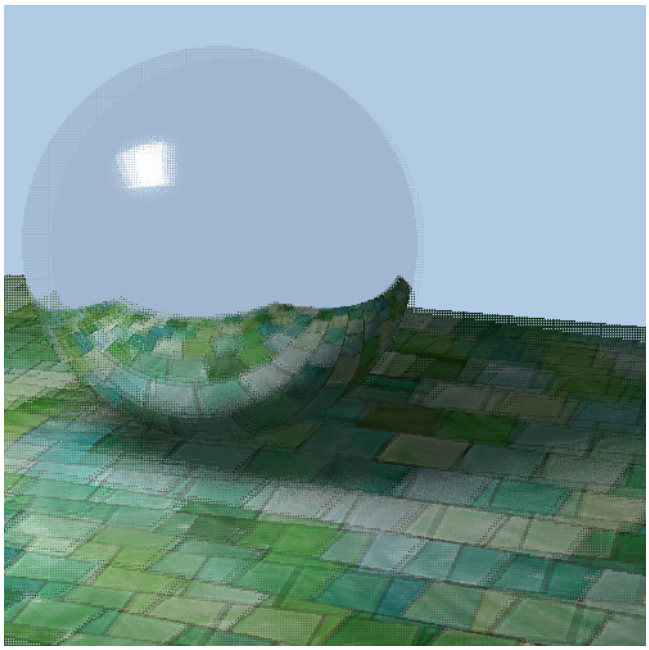

**Figure 13.** Blur from lack of color distance epsilon when interpolating

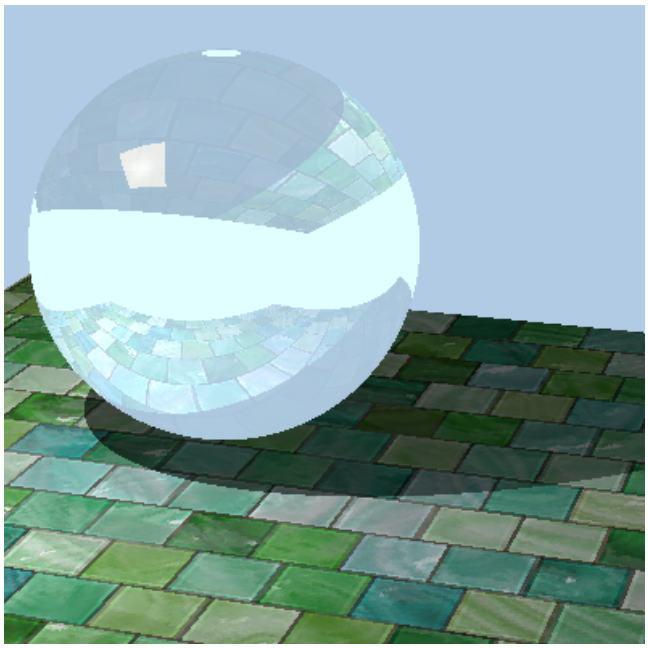

**Figure 16.** Incorrect refraction of rays through glass ball

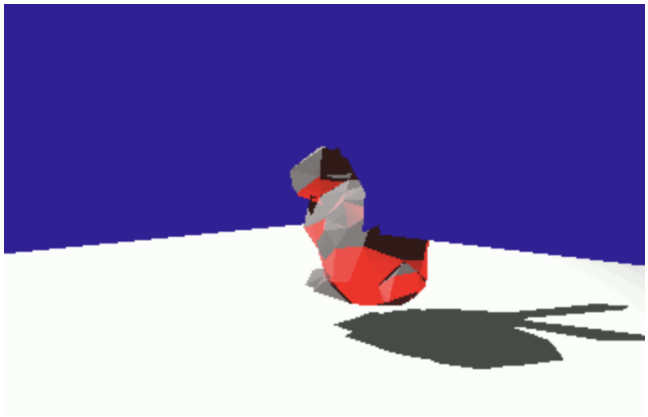

**Figure 17.** Deformed bunny due to extensively large panning motion without sufficient sampling

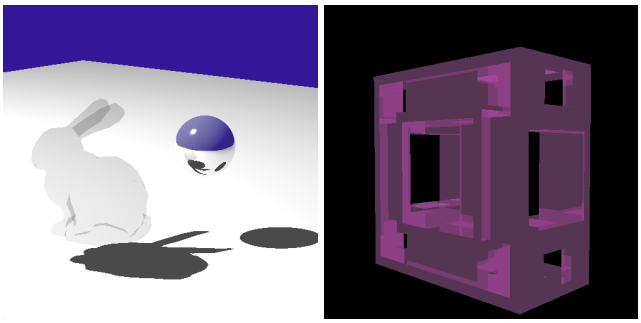

**Figure 14.** Triangle intersection with reversed normals

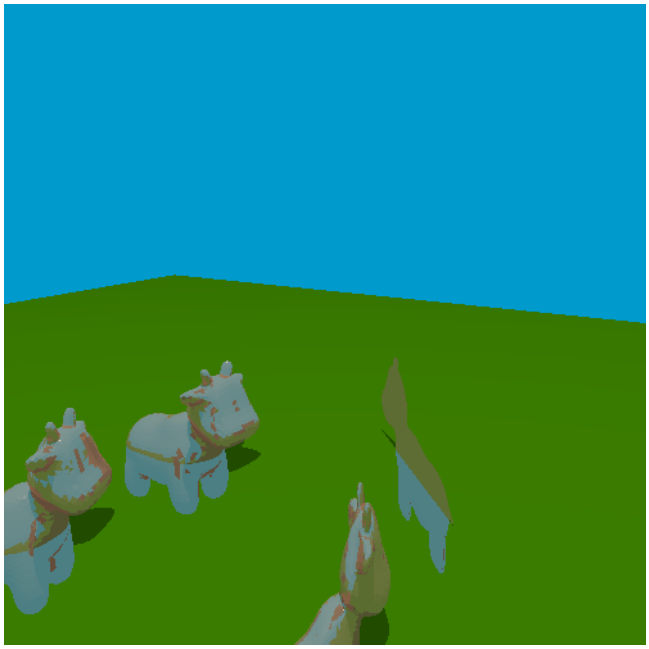

**Figure 15.** Error in algebraic transformation of mesh location when rotating objects