Ray Tracing and Beyond

JAMIE HONG, ANZE LIU, AKSHIT DEWAN, and TIM TU

ACM Reference Format:

Jamie Hong, Anze Liu, Akshit Dewan, and Tim Tu. 2023. Ray Tracing and Beyond. 1, 1 (May 2023), 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

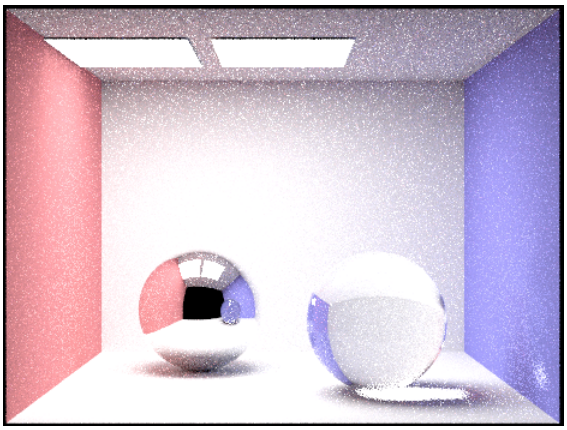


Fig. 1. Spheres with 2 lights, using photon mapping for caustics

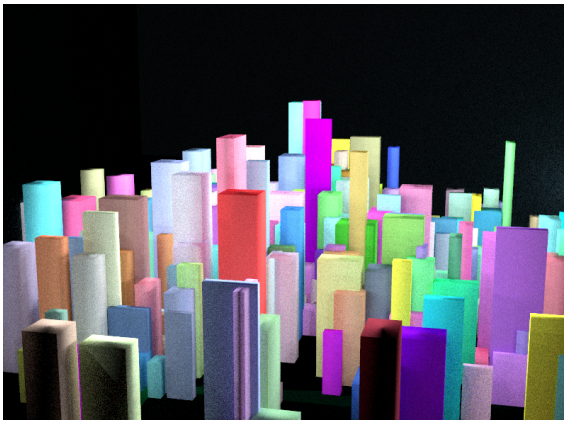


Fig. 2. Tiny city scene with 300 lights using k-d tree for cache points lookup

1 ABSTRACT

Ray tracing is a powerful rendering technique as it is based on simulation of the physical behavior of light. However, using ray tracing for rendering caustics for refractive objects or for scenes with thousands of lights can be inefficient.

To address these issues, we designed and implemented additional optimizations and features from scratch on top of a basic ray tracer. We chose to focus on two main additions: photon mapping, and a cache points optimization for lighting, inspired by Disney's Hyperion Renderer [1].

Along the way, this project has presented challenges in various aspects. For instance, we needed to consider how to integrate new techniques into the existing ray tracer, how to efficiently optimize our implementation, and how to best demonstrate our ray tracer's ability to render complex scenes: ones with clearer caustics or others with hundreds to thousands of lights.

With our objective and the project's challenges in mind, we augmented the basic ray tracer with photon mapping and cache point optimizations for lighting, basing our design off of the papers found in our references.

Authors' address: Jamie Hong; Anze Liu; Akshit Dewan; Tim Tu.

Manuscript submitted to ACM

2 TECHNICAL APPROACH

2.1 Photon Mapping

For photon mapping, we simulate photons by casting light rays into the scene, with the goal of being able to more accurately render caustics. At ray-object intersections, we store the information about the photon at this intersection, such as its power and direction, into a map. This occurs during a pre-rendering pass. Then, at render time, we use the nearest few photons in our map to estimate the radiance for a shading point. For details of the implementation, we referred to the relevant section from "A Practical Guide to Global Illumination using Photon Mapping" [4].

Comparison between Ray Tracing and Photon Mapping:

Ray Tracing	Photon Mapping
Trace rays from camera	Trace photons from light source
Rays propagate radiance	Photons propagate flux
Requires many rays and bounces for caustics	Dense photon map where caustics are located

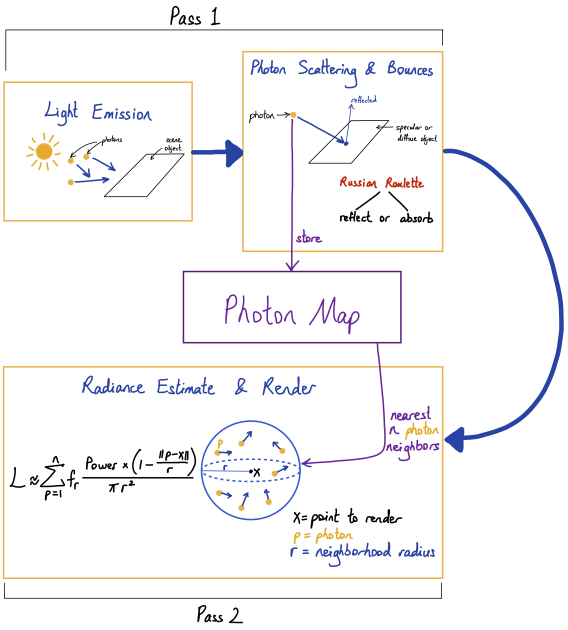
Our two-pass photon mapping algorithm is as follows:

Pass 1: Construction of global photon map

(1) Send rays from each light source into the scene to represent the travel of photons.
(2) Check for ray-object intersections.
(3) Use Russian Roulette to decide if a photon is reflected or absorbed at point of intersection.
(4) The photon at this point of intersection is stored in a k-d tree.

Pass 2: Radiance estimation using photon map

(1) Based on the neighborhood radius, obtain a list of photons that are neighbors to the point to be rendered using k-d tree lookup.
(2) Compute radiance from each nearby photon using the point's BSDF function and the photon's power, scaled by the distance between the photon and point.
(3) Sum the radiance estimates from the neighboring photons together, as resulting radiance.
(4) Estimate radiance for every point in scene.



We tuned the following parameters to achieve the final rendered results.

(1) Number of rays sent into the scene to construct the photon map.
(2) Radius within which photons are used to estimate the radiance at a particular intersection point of interest.

The more rays we send into the scene or more photons we use to estimate radiance, the longer the scene takes to render. Therefore, tuning the number of light rays and the neighborhood radius to achieve efficient processing while still utilizing the photon map to render caustics is a challenge.

In addition, in the original photon mapping paper [4], three photon maps are constructed: caustic, global, and volume photon maps. In our project, we implemented only the global photon map to achieve an approximate representation of the global illumination for the scene while still keeping the render time reasonable.

2.2 Cache Points Optimization

In path tracing with importance sampling, we loop over every light source to compute the radiance estimate for each point. However, considering every light becomes more expensive as the number of light in the scene increases, especially if some of the lights far away from the shading point have negligible contribution or are occluded.

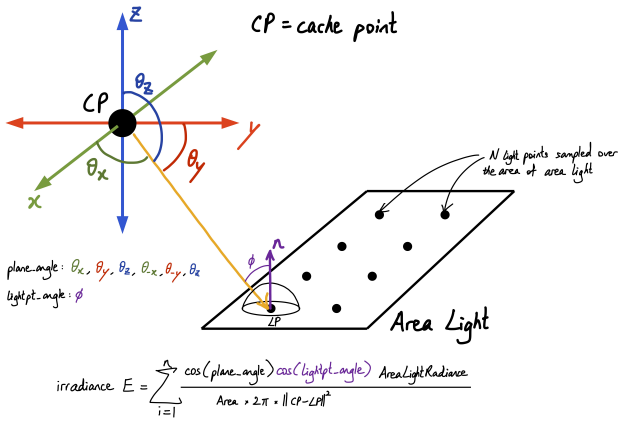
In order to render scenes with hundreds to thousands of lights, we would like to perform efficient light sampling. One method of doing so is to "cache" region-specific lighting information into a certain number of cache points. From these, we can then use the nearest cache point to sample light for a shading point, resulting in more efficient computation during the rendering phase.

As for the basis of this idea, "The Design and Evolution of Disney's Hyperion Renderer" [1] introduces cache points optimization for their goal of rendering scenes with hundreds of thousands of lights, such as in ones in the movie *Big Hero 6*. We also referred to Pixar's "Production Volume Rendering" [2], for additional subtleties in the design.

Our final design consisted of a 2-stage cache point implementation:

Stage 1: Pre-compute phase: building cache points

(1) Initialize uniformly randomly distributed cache points within bounding boxes.
(2) For each cache point, create 8 light lists, each represents a discrete distribution over light sources, based on their incident irradiance and visibility (calculated by sampling rays from lights).
  (a) 6 lists of lights and irradiances for the cardinal planes (1 list for each directional plane).
  (b) 1 list of lights and irradiances for the omnidirectional receiver.
  (c) 1 list of lights which are very close to the cache point, to account for rapid changes in illumination between cache points.
(3) Filter out insignificant lights by only storing the most significant lights until 97% of total irradiance at this cache point is reached.
(4) The cache points are stored in a k-d tree.



Incident irradiance from area light to cache point.

Stage 2: Light sampling and aggregating phase: using cache points

(1) For each point of interest, sample light from the closest cache point using the k-d tree, which returns the nearest neighbor [3].
(2) Randomly choose a distribution from cardinal, omnidirectional, and nearby light distributions, by weighting each distribution by the dot product of its normal (when applicable) with the surface normal of shading point.
(3) Randomly sample a light from the chosen discrete distribution, weighting by the visibility of the light.

(4) Use the chosen light to generate more samples for estimating radiance at a point of intersection. Sampling from lights in this way allows us to use cache points for importance sampling.

Parameters tuned to achieve quality render while lowering computation time:

(1) Number of cache points (per bounding box)
(2) Number of lights used for cache sampling
(3) Number of samples per pixel

As the number of cache points, lights, and samples per pixel increases, the quality of the render improves. The higher the density of cache points, the more accurately the nearest cache point can approximate the relevant lighting at a shading point. However, improved accuracy comes at the cost of increased render time for a scene, as lookup time could increase with more cache points.

2.3 Challenges and Resolutions

2.3.1 Scenes and Procedural Generation. In the process of implementing our optimizations, we realized that the original scenes from the project were likely insufficient to demonstrate the results of our optimizations. We approached this in two main ways.

One method we took was to start from the existing `.dae` files included in the project. We updated the existing scene to change the setup of the lights from 1 large area light, to 2 lights, and then to 440 smaller lights.

We were also interested in producing our own scenes, and looked into generating Blender scenes from scratch using Blender's Python interface. Once we had written a script for procedurally generating city scenes, with many buildings (cubes) and many lights (area lights), we could then use the script to generate various `.dae` files that we could then use to render effects such as caustics or demonstrate the efficiency of our renderer with a large amount of lights.

We created approximately 5 city scenes, with the number of lights ranging from 300 to 5000 lights. With a larger number of lights in each scene, we were able to better observe improvements in render time for our cache point optimized renderer in comparison to the original basic ray tracer implementation.

2.3.2 Data Structure for Cache Points Lookup. We experimented with various storage methods for efficient neighbor lookup of cache points. After profiling the rendering times and comparing the resulting renders of the different methods, we decided to use a k-d tree, which enabled shorter render times with minimal artifacts.

Vector	K-D Tree	Spatial Map
Expensive sort	Optimize cache point lookup with few artifacts	Checkered pattern artifacts (10)

2.4 Lessons Learned

Over the course of this project, we learned how to interpret and transfer methods described in research papers to fully functional implementations, under the constraints of limited resources and time.

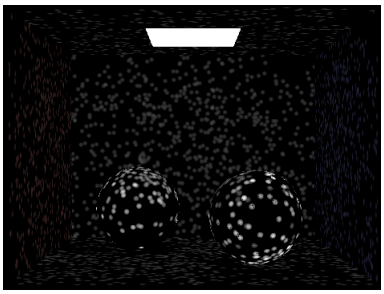
For example, we extracted the key steps of the photon mapping algorithm: constructing the photon map, Russian Roulette for photon scattering, and radiance estimation. We determined that cone filtering, Gaussian filtering, and caustic and volume photon maps would be useful for further improving render results, but were not key for our fundamental photon mapping implementation.

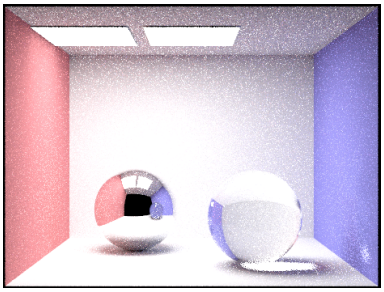
We utilized Blender and its Python API to procedurally generate city models with buildings of refractive or reflective material, and illuminated by area lights randomly dispersed throughout the scene, for demonstrating our cache point optimization results.

Finally, we learned how different parameters can introduce tradeoffs between render time and accuracy, and how to optimize our program to balance reducing computation time while maintaining high-quality renders.

3 RESULTS

3.1 Rendered Scenes with Photon Mapping





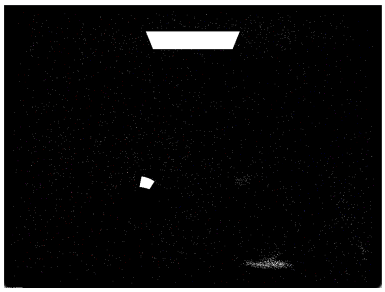
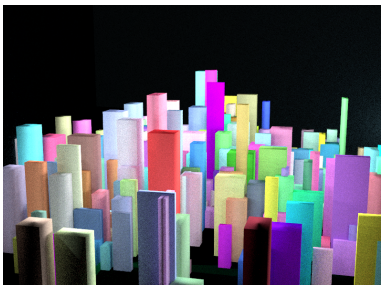


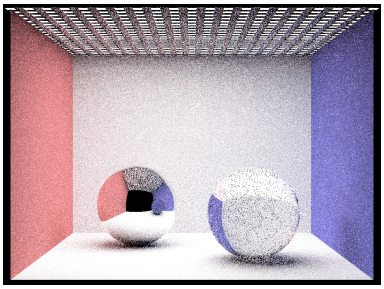
Fig. 3. Spheres - photon mapping with one bounce radiance

Fig. 4. Spheres - photon mapping with six bounce radiance and cache point optimization

Fig. 5. Spheres - photon map

3.2 Rendered Scenes with Cache Points





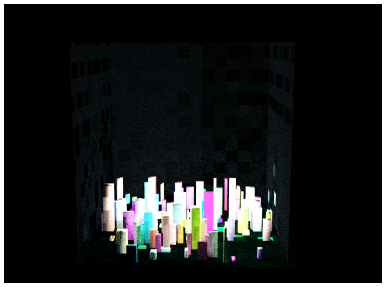
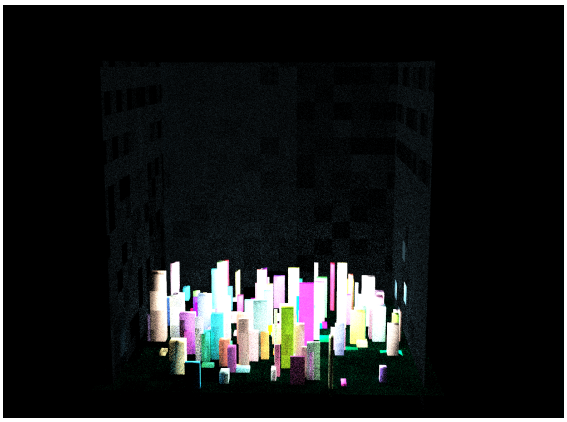


Fig. 6. Tiny city scene with 300 lights using kd tree for cache points lookup

Fig. 7. Spheres with 440 lights using k-d tree for cache points lookup

Fig. 8. Tiny city scene with 5000 lights using spatial map for cache points lookup

3.3 Speedup using Cache Point Optimization



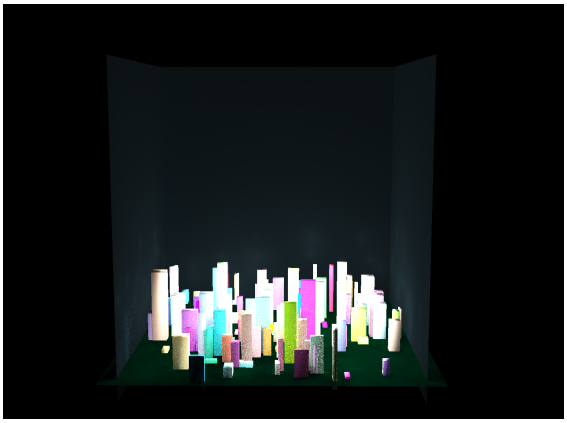


Fig. 9. Tiny city scene with 5000 lights using spatial map for cache points lookup (render time: 115.59 s)

Fig. 10. Tiny city scene with 5000 lights using staff renderer for comparison (render time: 375.49 s)

4 CONCLUSION

Efficiently rendering complex scenes, such as those with a lot of caustics and light sources, requires complex optimizations. We have explored and implemented two such optimizations: photon mapping and cache points. Yet ultimately, we are only making approximations of the physical behavior of lighting through these methods. As a result, our optimizations may trade off image quality or low amounts of noise to achieve faster rendering times. From this project onwards, future optimizations can be further explored to expand our renderer's ability to that of production level renderers such as Disney's Hyperion, and to allow us to render even more exciting scenes than before!

5 CONTRIBUTIONS FROM EACH TEAM MEMBER

(1) Jamie: pseudocode and implementation for cache point and photon mapping, structure used to store cache points (kdtree), Blender script for procedural scene generation, tuned and tested with different parameters, final video, final report
(2) Tim: implementation of photon mapping, cache point construction, structure used to store cache points (spatial map), light sampling, generated .dae files to be rendered, final video
(3) Akshit: implementation of photon mapping construction, cache point light sampling, generated .dae files to be rendered, tuned and tested with different parameters, final video
(4) Anze: pseudocode for cache point and photon mapping, implementation for cache points construction and light sampling, milestone report, final video, slides, final report

REFERENCES

[1] Brent Burley, David Adler, Matt Jen-Yuan Chiang, Hank Driskill, Ralf Habel, Patrick Kelly, Peter Kutz, Yining Karl Li, and Daniel Teece. 2018. The Design and Evolution of Disney's Hyperion Renderer. *ACM Trans.Graph.* 37, 3 (July 2018). https://doi.org/10.1145/3182159
[2] Julian Fong, Magnus Wrenninge, Christopher Kulla, and Ralf Habel. 2017. Production Volume Rendering: SIGGRAPH 2017 Course. In *ACM SIGGRAPH 2017 Courses* (Los Angeles, California) *(SIGGRAPH '17)*. Association for Computing Machinery, New York, NY, USA, Article 2, 79 pages. https://doi.org/10.1145/3084873.3084907
[3] J. Frederico Carvalho 2008. kdtree implementation. https://github.com/crvs/KDTree.
[4] Henrik Wann Jensen. 2002. A Practical Guide to Global Illumination using Photon Mapping. *Siggraph 2002 Course 43* (July 2002).