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

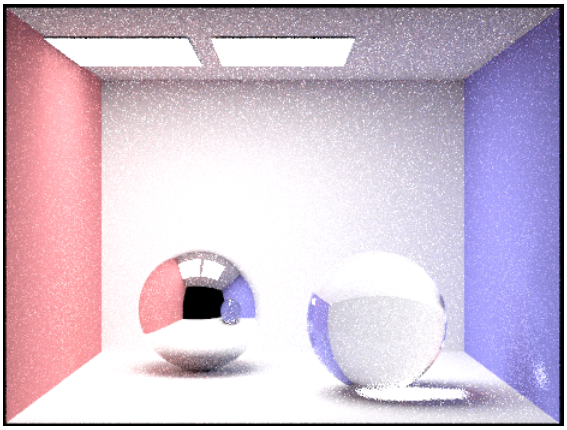


Fig. 1. Spheres with 2 lights, using photon mapping for caustics

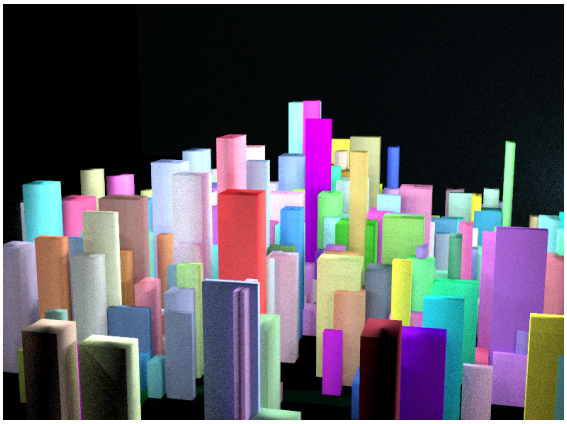


Fig. 2. Tiny city scene with 300 lights using k-d tree for cache points lookup

1 ABSTRACT

Ray tracing is a powerful rendering technique as it is based on simulation of the physical behavior of light. However, it can also be incorrect when rendering caustics for refractive objects and inefficient, especially when sending out many rays from camera into the scene to render or when importance sampling **all** lights in scene to compute radiance. To address these issues, we begin with a ray tracer and then implement from scratch optimizations and features inspired by Disney's Hyperion Renderer [1]. This project is challenging in various aspects, for instance, how to integrate new techniques into the existing ray tracer, how to efficiently optimize our ray tracer, and how to best demonstrate our ray tracer's ability to render complex scenes with larger number of light sources. With our objective and the challenges in mind, we augmented the basic ray tracer with photon mapping and cache point optimizations for lighting based upon our references.

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2023 Association for Computing Machinery.
Manuscript submitted to ACM

2 TECHNICAL APPROACH

2.1 Photon Mapping

In photon mapping, we first cast light rays into the scene. At ray-object intersections, we store information about the photon at this intersection into a map. This happens in a pre-rendering step. Then, at render time, we use the nearest few photons from our map to estimate the radiance for a shading point. We implemented photon mapping based on the relevant section from "A Practical Guide to Global Illumination using Photon Mapping" [4].

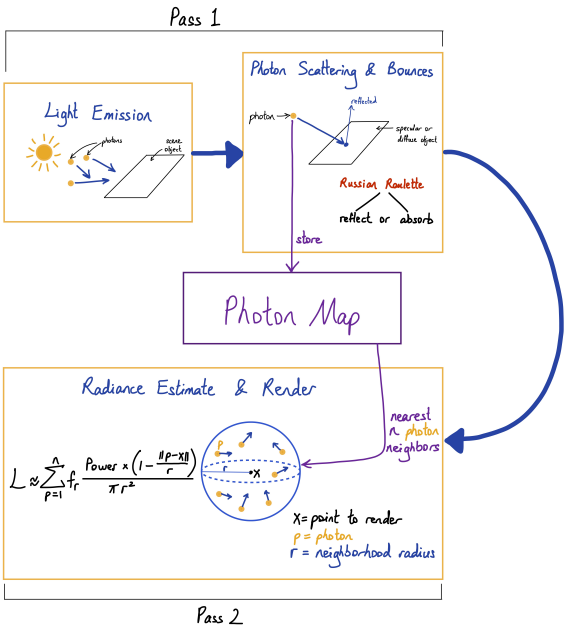
Comparison between Ray Tracing and Photon Mapping:

Ray Tracing	**Photon Mapping**
trace rays from camera	trace photons from light source
rays propagate radiance	photons propagate flux
cannot correctly generate caustics	generate caustics for refractive objects

We concluded the a 2-pass photon mapping algorithm as follows:

(1) Pass 1: Construction of global photon map
   (a) Send rays from each light source into the scene to represent the travel of photons.
   (b) Check for ray-object intersections.
   (c) Use Russian Roulette to decide if a photon is reflected or absorbed at point of intersection.
   (d) The photon is added to a k-d tree.

(2) Pass 2: Radiance estimation using photon map
   (a) Based on the neighborhood radius, obtain a list of photons that are neighbors to the point to be rendered using k-d tree lookup
   (b) Compute radiance from each nearby photon using BSDF function and photon power scaled by distance between photon and the point
   (c) Sum the radiance estimates from every pair of photon-shading point together
   (d) Use the radiance estimate to render every point in scene



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

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

The more photons we use to estimate radiance or the more rays sent into the scene, the longer the computation time, which is against the objective of this project. Therefore, tuning the radius and number of camera rays to achieve efficient processing while rendering caustics is challenging. In addition, in the original paper on photon mapping, three photon maps are constructed: caustic, global, and volume photon maps. We implemented only the global photon map to achieve an approximate representation of the global illumination for the scene so that we are still able to render within reasonable time.

2.2 Cache Points Optimization

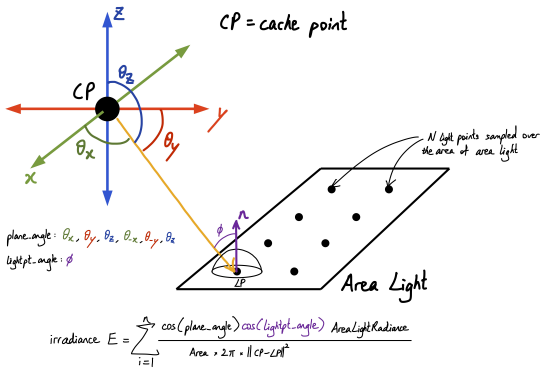
In order to render scenes with many complex light sources, we need to perform efficient light sampling by approximating incident radiance. To render a point in scene, we provide the information about the important lights for the point's position and normal such that the radiance can be estimated from this information. Cache Point optimization used in the paper, "The Design and Evolution of Disney's Hyperion Renderer" and Pixar's "Production Volume Rendering" stores this information to be used for a shading point [1] [2] .

In path tracing with importance sampling, we loop over every light source to compute the radiance estimate for each point, which is expensive, especially considering the negligible contribution of far away or occluded lights. Instead, we can "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 point, resulting in more efficient computation during the rendering phase.

We concluded a 2-stage cache point implementation:

Stage 1: Pre-compute phase: building cache points

(1) Initialize certain number of randomly distributed cache points within bounding boxes
(2) For each cache point, create a total of 8 lists of light, each representing a discrete distribution over light sources, and compute their irradiance
   (a) 6 lists of lights and irradiances for the cardinal planes (1 list for each plane/direction).
   (b) 1 list of lights and irradiances for the omnidirectional receiver.
   (c) 1 list of lights that are very close to the cache point, to account for rapid changes in illumination between cache points.



Example: 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 points using kdtree, which returns the nearest neighbor [3].
(2) Choose a distribution based on the weights for cardinal, omnidirectional, and nearby distributions, which depend on the surface normal of the shading point.
(3) Randomly sample a light from the chosen discrete distribution.
(4) Use the chosen light to generate more samples for estimating radiance at a point of intersection.

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, the time taken to render a scene increases while the quality if render improves. For instance, the cache points are specific positions in scenes that store information about the lights for that position. The more cache points there are, the more accurately the light sampling from nearest cache point can represent the physical behavior of lighting at a shading point. However, more cache points takes longer to construct and lookup. After parameter tuning, our best implementation used 120 cache points to render Tiny City scene in Fig.6.

2.3 Challenges and Resolutions

(1) In the process of implementing our optimizations, we realized that the scenes from the project were likely insufficient to demonstrate the results of optimizations. We approached this in two main ways. First, we started from the `.dae` files included in the project, and changed the setup of the lights from 1 large area light to 2 and then to 440 smaller lights. Next, we used blender python script to generate models with different sizes, reflective or refractive material, and light sources. We created two Tiny City scenes, one with 300 lights and another with 5000 lights. The more lights there are, the more difference in render time we can observe between our cache point implementation and staff ray tracer solution.

(2) To optimize cache points, we experimented with how to store cache points for efficient neighbor lookup. We eventually used K-D Tree because it enables shorter render time with low artifacts.

Vector	K-D Tree	Spatial Map
Expensive sort	Optimize cache point lookup with few artifacts	Checkered pattern artifacts (see fig.11)

2.4 Lessons Learned

Over the course of this project, we learned how to interpret and transfer the methods described in research papers to actual implementation given limited resources and time. For example, we extracted the most important steps in the algorithm described in the photon mapping paper: use Russian Roulette for photon scattering, constructing photon map, and radiance estimate, while removing the cone filtering and Gaussian filtering, which are used to further improved render results. We also learned to use blender and python script to generate city scenes with random-size building blocks, refractive and reflective material, and area lights. Finally, we learned how different parameters influence the render result and how to optimize our program to reduce computation time.

3 RESULTS

3.1 Rendered Scenes with Photon Mapping

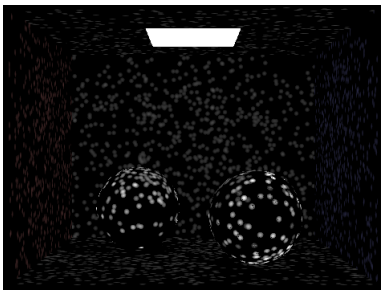


Fig. 3. Spheres - photon mapping with one bounce radiance

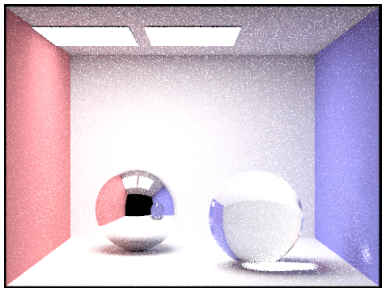


Fig. 4. Spheres - photon mapping with six bounce radiance and cache point optimization

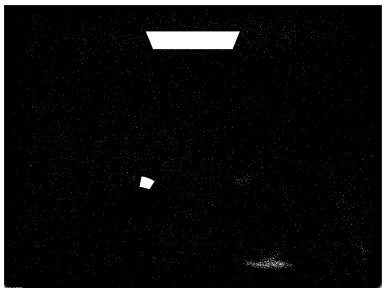


Fig. 5. Spheres - photon map

3.2 Rendered Scenes with Cache Point

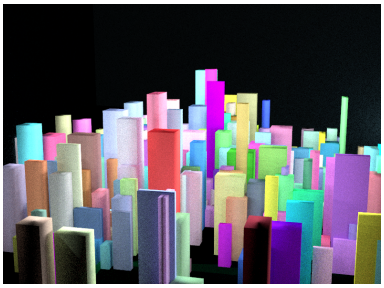


Fig. 6. Tiny city scene with 300 lights using kd tree for cache points lookup

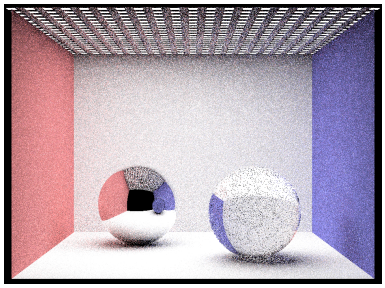


Fig. 7. Spheres with 440 lights using k-d tree for cache points lookup

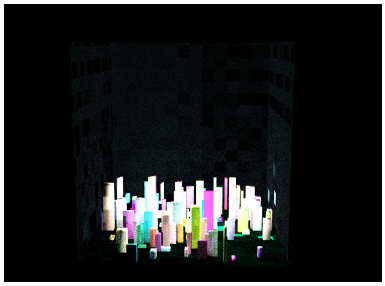


Fig. 8. Tiny city scene with 5000 lights using spatial map for cache points lookup

3.3 Speedup using Cache Point Optimization

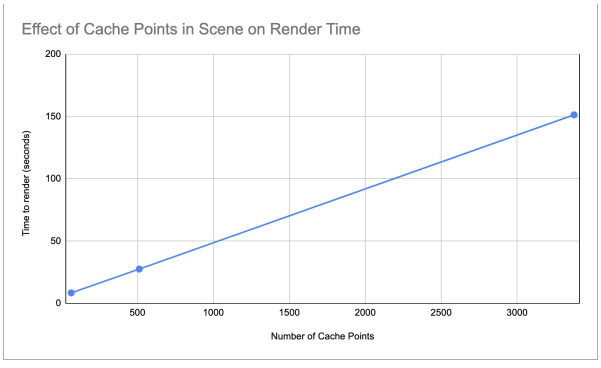


Fig. 9. Plot showing as the number of cache points increases, the render time increases.

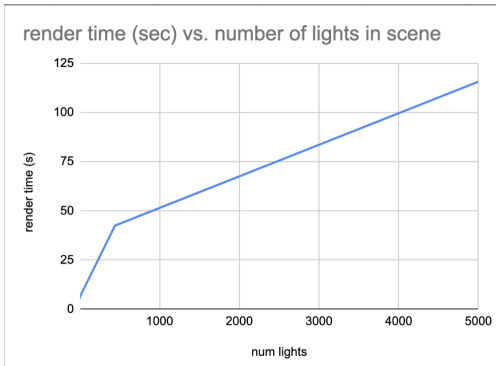
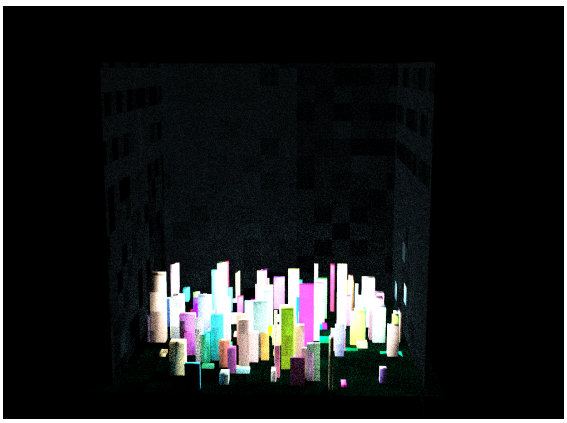


Fig. 10. Plot showing as the number of lights in scene increases, the render time increases



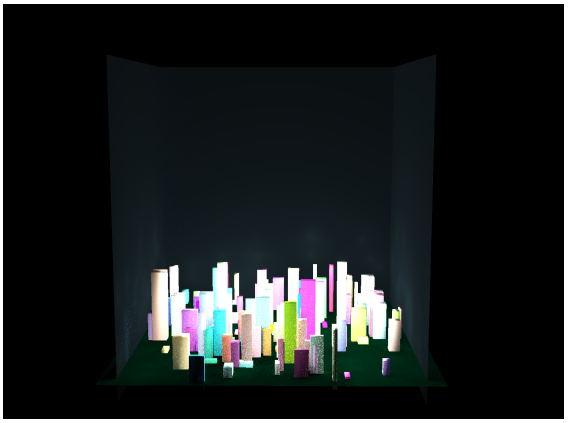


Fig. 11. Tiny city scene with 5000 lights using spatial map for cache points lookup

Fig. 12. Tiny city scene with 5000 lights using staff renderer for comparison

4 CONCLUSION

Rendering complex scenes within reasonable time requires complex optimizations. We have covered two such optimizations, which are still approximations of the physical behavior of lighting. As a result, our optimizations may tradeoff image quality or noise for the sake of efficient rendering. From this project onwards, the optimizations can be further explored, to better approach the rendering ability of production level renderers such as 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), generated .dae files to be rendered, tuned and tested with different parameters
(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
(3) Akshit: implementation of photon mapping construction, cache point light sampling, generated .dae files to be rendered, tuned and tested with different parameters
(4) Anze: pseudocode for cache point and photon mapping, implementation for cache points construction and light sampling, milestone and final report, slides

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 [n. d.]. kdtree implementation, note = https://github.com/crvs/KDTree, year = 2008.

[4] Henrik Wann Jensen. 2002. A Practical Guide to Global Illumination using Photon Mapping. *Siggraph 2002 Course 43* (July 2002).