Extending Photon Mapping

Greg Cowan
Rensselaer Polytechnic Institute
cowang@rpi.edu

Nate Wheeler
Rensselaer Polytechnic Institute
wheeln@rpi.edu

ABSTRACT

In our work on our final project for Advanced Computer Graphics class, we implemented a selection of improvements to Homework 3's Ray Tracing and Photon Mapping. We focused on speeding up three parts: the photon casting, the ray tracing, and the photon gathering. An Irradiance Cache was used to intelligently place and sample photons. The volumetric data structure for storing photons, a kd-tree, is now balanced on insertion, to make photon gathering faster. Fewer shadow rays are cast, by using shadow photons. And, we present a planned feature: antiphotons, which parallelize photon casting by localizing the visibility computation.

KEYWORDS

Photon Mapping, Ray Tracing, Parallelization, KD-Tree, Shadow Photons, Antiphotons

ACM Reference format:
Greg Cowan and Nate Wheeler. 2017. Extending Photon Mapping. In *Proceedings of Advanced Computer Graphics, , (Spring 17),* 7 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Advanced Computer Graphics class' Homework 3 was a great hands-on implementation of the core algorithms of a ray tracer and photon mapper. In this paper, we detail our final project's extensions to Homework 3. We dove deeper into more efficient - though more complicated - algorithms for those rendering topics. We were particularly focused on improving the photon mapping portion. The kd-tree volumetric data structure for storing photons was unbalanced in the initial homework 3 implementation, but we showed that balancing it can speedup lookup of photons during the gathering step. Many shadow rays are needed to be traced from geometry to the light source to produce sufficiently soft shadows in a ray tracer, especially in scenes with large area lights. We show how an implementation of shadow photons can significantly reduce the number of shadow rays, without sacrificing the quality of shadows, by informing the ray tracer about whether a hit point is directly illuminated, in the penumbra, or completely shadowed. We attempted to reduce the number of photons needed, by starting an implementation of an irradiance cache. We also explored the idea of using antiphotons to decouple photon casting from the global visibility calculation, inspired by antiradiance. We discuss the benefits and drawbacks of this idea when applied to photons. Also, we will discuss an idea to speed up photon gathering from the kd-tree using a k-nearest-neighbors search algorithm, instead of the provided radius query.

Spring 17, ,
2017.

2 RELATED WORKS

Our implementations were based on or inspired by algorithms from research papers and presented in class. And, as mentioned, we started with Homework 3 code from class.

Photon Mapping

Photon Mapping is a technique that was developed by Henrik Wann Jensen in his set of 1996 papers [2]. Effectively, it traces a bunch of rays in random directions from a light source and stores where they hit and how much energy they have. These "hits" are labeled photons and they can accurately describe how light would realistically bounce around the scene. Then modified ray tracing occurs, where upon determining a hit, photon are then gathered from near the hit point and used to determine global lighting, indirect lighting, and caustic effects.

Shadow Photons

"Efficiently Rendering Shadows Using the Photon Map" by Jensen and Christensen [3] introduces the idea of shadow photons. Shadow photons are intended to alleviate the cost of casting a large number of shadow samples by identifying which areas of an image are completely illuminated or completely in shadow, and not casting shadow samples for those areas. This significantly cuts down on the number of shadow rays, drastically improving raytracing performance at only a small cost to the photon tracing and gathering steps. It is especially valuable when using a large number of shadow samples.

Irradiance Cache

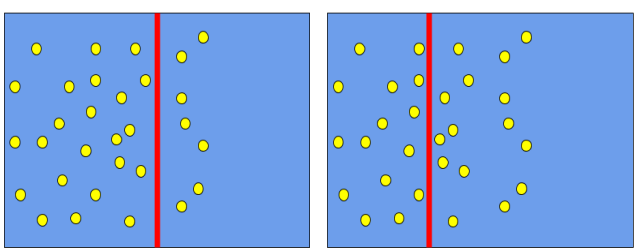
In Ward's "Ray Tracing Solution for Diffuse Interreflection," [4] he introduces the idea of not placing down every photon. Since diffuse reflection changes very slowly and smoothly, he proposes that it should be a good approximation if some photons with similar attributes (normal, energy, etc) are averaged into one entity. These leftover sample points are then used to interpolate the indirect lighting in the scene. This helps to reduce memory usage and render time since there are fewer photon objects available.

Antiradiance

"Implicit Visibility and Antiradiance for Interactive Global Illumination" by Dachsbacher et. al. [1] shows how to turn radiosity into a local computation instead of a global one, by eliminating the visibility calculation. They do this by making visibility implicit and correcting with another propagated value *antiradiance*. Antiradiance is negative radiance that propagates from a hit point of radiance in order to cancel out the extra light that was placed on objects that should be occluded from the light source. The method allows parallelization, so the authors implemented a GPU based approach. We'll take the ideas of antiradiance and apply them to photon mapping.

3 ALGORITHMS

3.1 Balancing the KD Tree



a. *b.*

Figure 1: Comparison of divide algorithms. Split size of 30

One simple change we wanted to make from the beginning was finding a more efficient way of handling the kdtree. In its original state, when cells had enough photons to split, they were naively divided based on the midpoint of the longest axis of the cell. This process left many cells unbalanced, especially in scenes involving caustics due to how quickly areas go from sparse to dense (Figure 1a). In order to address this, we came up with a solution to make sure cells were split evenly every time. Upon reaching maximum capacity, a cell still picks the longest axis to split on. However, instead of dividing across the midpoint, the set of photons in cell is sorted based on the selected dimension. It then is split based on the average of the two medians when splitting using an even number (Figure 1b). Note that an average is taken between the median one greater than the median in an odd number case in order to make sure there are no photons on the dividing line. This optimization provides a small, yet still noticeable increase in performance when doing the gather of photons, especially when a large number of photons was used. In fact, while the tracing of photons took slightly longer when using a small number of photons, it actually began to get faster due to the shorter traversal time. See Table 4.

Table 1: Time Comparisons with New KD Tree

Test Type	Trace Time	Render Time
10,000 Photons, 4 Shadow Samples		
Old KD Tree:	.2 s	44 s
New KD Tree:	.3 s	30 s
500,000 Photons, 4 Shadow Samples		
Old KD Tree:	8 s	48 s
New KD Tree:	7 s	32 s

3.2 Shadow Photons

In our work we implemented shadow photons as described by Jensen and Christensen in "Efficiently Rendering Shadow Using the Photon Map" [3]. Shadow photons are designed to reduce where shadow rays must be cast from. Instead of casting many jittered shadow samples to the light source on every hit of the final rendering when ray tracing, the photon gathering tells the renderer when it needs to cast shadow rays.

The first modification to a traditional photon mapping rendering scheme is in the photon casting. Normally, photons are cast from the light source, out a random direction, and intersection is tested with all the geometry and placed at the closest point to the light that it hits, then the algorithm recurses in a new specular or diffuse bounce direction. That algorithm models the real-world behavior of photons well, but shadow photons aren't inspired by the real world, but rather are an imaginary extension that allows for useful information later in the rendering pipeline. Shadow photons are placed at all the other hits with geometry, after the one closest to the light source. They are visualized in blue in Figure 2. This has the effect of placing shadow photons on the back surface of objects, and on the ground in the shadow of the object. To simplify our implementation, once the closest hit has been returned, we just recursively trace a new zero-energy photon in the same direction as the current illumination photon. A better method would be to overhaul the collision code in RayTracer::CastRay – which we also use to cast photons – to place shadow photons on all but the closest hit. Our naive recursive method is $O(n*h)$ where n is the number of geometry objects in the scene and h is the number of hits in a given direction from the light source with geometry in the scene. But, the better method would be $O(n)$, since the illumination photon and the shadow photons are all placed in one pass. Then, in the photon gathering step, photons are classified into three different categories. Ones that have zero energy are shadow photons, ones that were the first iteration or the ones directly from the light are classified as direct illumination photons, and lastly the remaining photons that represent light after multiple bounces are classified as indirect

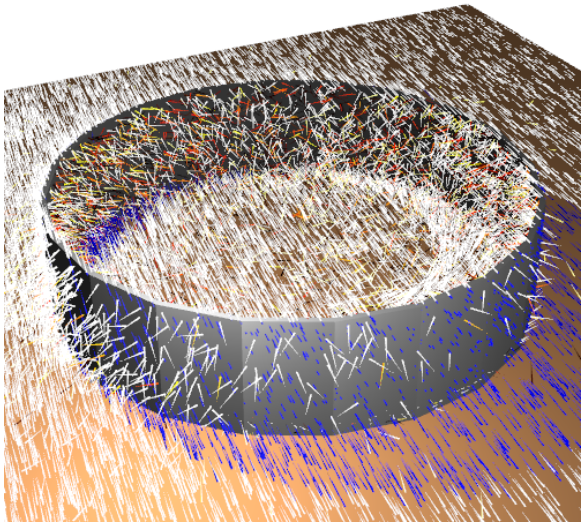


Figure 2: Shadow photons are visualized in blue

Figure 3: Blue where shadow rays are cast

photons. For rendering of global illumination and caustics, both direct and indirect are used as usual. But, for the shadow photon algorithm, the number of shadow photons and direct illumination photons are important. The counts are sent off to the ray tracer, which uses them to make a decision about whether to cast shadow rays or not.

Finally, the ray tracer compares the counts of shadow and direct photons. There are three cases. If the number of shadow photons is zero, then the hit point is directly illuminated by the light source and the contribution of the light is calculated and added to the color of the pixel – without having to trace shadow rays. If the number of direct illumination photons is zero, then the hit point is completely in shadow, so no direct contribution from the light source is added (indirect light may be added by the photon gathering later). Lastly, if neither of the counts are zero, then that indicates that the hit point is in or near a penumbra of a soft shadow, so shadow rays are cast to the light source, like in a standard ray tracer. The hit points that traced shadow rays are tinted blue in Figure 3.

Some special points about our implementation follow. We store shadow photons with zero energy, rather than negative energy. Negative energy could be used if the implementation used the photon map directly to calculate shadow lighting, thereby eliminating the need for any shadow rays at all. But, the Jensen and Christensen paper [3] shows how that method results in overly soft shadows, so we skipped it in our implementation. Plus, that can have other artifacts along corners and meeting points of geometry, unless the photon gathering implementation is very robust. Ours is not, which is why shadow rays are cast from the edge of the ring that is in shadow in the above figure, because direct illumination photons are picked up from the other side of the ring. Also, it is possible to extend this method to work for multiple light sources, by storing which light source each photon is from and counting and adding / not adding the contributions from each light source independently.

Figure 4: Blue where shadow rays are cast

We didn't implement that feature, though the reduction in the number of shadow rays would most likely be multiplied by the number of light sources. A limitation of this approach is that it needs direct photons to be stored, and it needs the extra shadow photons to be stored. The direct illumination photons increase the size of the memory required for the kd-tree to store them. And, the shadow photons can add a lot of time and memory usage in complex scenes with many shadows. The Jensen paper gives the example of leaves on a tree, as an example likely to have lots of shadow photons. However, it is possible to use the ideas shadow photons without storing

direct illumination photons. This is visualized in Figure 4. The only modification here is that direct illumination photons aren't stored, and that they therefore can't be counted, and so direct illumination areas can't be detected. Except that, we can assume that if the count of the number of shadow photons is low enough compared to the number of photons collected, then the point is directly illuminated, since the query radius will increase to include enough indirect photons. This results in an imperfect shadow photon implementation, but in practice is a huge decrease in memory usage – since there are no direct illumination photons being placed and there are a large proportion of them in our scene – and with a low enough proportion of shadow photons to indirect illumination photons, there are no visual artifacts in the shadows, just more shadow samples. For our test scenes, we found that when the count of shadow photons was less than 1/4th the number of photons collected, that there aren't artifacts, yet the number of shadow samples isn't too high. This method works best on scenes with lots of indirect lighting. Results for the reflective ring scene with 100,000 photons, 4 shadow samples, 1 antialias sample, 50 photons to collect, and size 500 by 500 are in Table 2.

Table 2: Shadow Photons on Reflective Ring, 4 Shadow Samples, $100,000$ photons

Type	Render Time	Shadow Rays
Full Shadow Photons:	39 s	465,342 shadow rays
No Direct Photons:	44 s	503,848 shadow rays
No Shadow Photons:	42 s	1,194,985 shadow rays

3.3 Irradiance Caching

Our version of irradiance caching is quite different than that proposed by Ward. We both use the same method of looking up valid photons in the nearby area and seeing if there is enough data to compensate for the missing photon. This process greatly reduces the scalability of photon tracing, since instead of simply storing a photon, it has to search for neighbors. However, this precomputation pays off in both using less memory and having shorter render times. If a potential photon has determined that there are enough valid neighbors to represent it, it then distributes its energy into its neighbors based on distance. This is where our method diverges from Ward's: in his paper, he averages the energy into one sample point whereas we simply sum up the energies. This allows us to get away without using an interpolation scheme in our gathering stage; however it means we have to be pickier about which photons are valid (more photons, smaller radius). This restriction means that our irradiance cache is most beneficial when dealing with a very large number of photons. In fact, it might slow down the system when using a small number, since the tracing process becomes much more expensive. Also, in order preserve caustics, we split the photon map into two structures: one for caustics and the other for general indirect lighting, much like Jensen does in his "Global Illumination" paper. Our irradiance caching scheme only applies to the general indirect map, as the caustics would be compromised otherwise.

Table 3: Comparisons with/without an Irradiance Cache

Test Type	Trace Time	Render Time	Memory
500,000 Photons, 4 Shadow Samples			
Without IC:	6 s	32 s	555 MB
With IC:	7 s	32 s	546 MB
5,000,000 Photons, 4 Shadow Samples			
Without IC:	56 s	28 s	3.15 GB
With IC:	80 s	28 s	1.48 GB

3.4 Parallelization

Our photon mapper and ray tracer both run on the CPU. Though individual parts are still serial, like visibility calculations, certain other parts need not be. For example, the photon mapper creates a new thread every time a new photon is generated. To avoid synchronization errors, the KDTree locks a node when a photon is being processed through it. Upon storing the photon, traversing further down the tree, or splitting the cell, the node's mutex is unlocked. This allows for a moderate speedup in photon tracing time. We also parallelized the ray tracer so that it could process multiple sets of pixels at a time. Since the ray tracer doesn't need to modify shared memory, no locks are needed and pixels can be calculated independently. We found that the easiest and most effective way to go about this was to give one thread a row of pixels to process, and the number of threads to be used before drawing can be specified by the user at launch. This development led to massive speedups, decreasing render time almost perfectly inversely with respect to the number of cores. We took advantage of C++'s std::thread library to accomplish both of these tasks.

Table 4: Time Comparisons with Multithreading

Test Type	Trace Time	Render Time
10,000 Photons, 4 Shadow Samples		
Serial:	.3 s	44 s
Multithreaded:	.2 s	30 s
500,000 Photons, 4 Shadow Samples		
Serial:	8 s	146 s
Multithreaded:	5 s	32 s

Note: These tests run with 9 threads on 8-core (4 physical, 4 logical) CPU.

3.5 Antiphotons

Antiphotons were inspired by the idea of Antiradiance as presented in "Implicit Visibility and Antiradiance for Interactive Global Illumination" by Dachsbacher et. al. [1] The idea of antiradiance is to allow the computation of radiance propagation to be massively parallelized on the GPU. It works by eliminating the need to perform visibility calculations for propagating radiance. That means that it just places the radiance on every surface that the traced ray

intersects with, not just the first one. That allows the hit detection to happen in parallel. Then, in order to account for the light that "went through" a solid object, a new anti-radiance is propagated in the same direction, with negative energy, to cancel out the light that shouldn't have gotten through the object. That can happen in parallel as well, which allows the implementation to be very quick on the GPU. The authors of the paper were successful in implementing just that. We thought about, and planned on implementing a similar idea for photon casting - called antiphotons. Which would be a natural extension of shadow photons. However, due to the complexity of implementing GPU code, the limited time, and since photon casting was already much faster than the ray tracing part of the algorithm, we decided to focus on other optimizations. A shadow photon implementation is different than an antiradiance one, because the memory requirements for propagating radiance and antiradiance are fixed; propagation just involves updating float values in an array. However, our photon implementation uses photon objects stored in a kd-tree in memory. So, a full antiphoton implementation would require a kd-tree on the GPU, thus complicating the shared memory of all the GPU cores, and adding overhead. A better idea than fully implementing antiphotons for our project would be to improve shadow photons so that they are placed in $O(n)$ time instead of $O(n * h)$ time as described in Section 3.2.

3.6 Triangle Meshes

We also implemented the ability to render and ray trace triangle meshes. So that scenes can use the bunny model. See Figure 5.

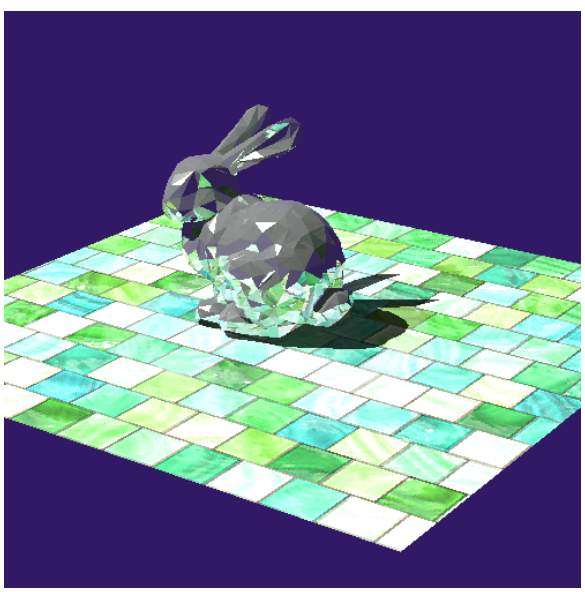


Figure 5: A ray traced reflective bunny with 1k triangles

4 CONCLUSIONS

Our work to improve the efficiency of the photon mapper and ray tracer was largely successful. We saw massive speedups to ray tracing from multi-threading and shadow photons. We produced faster photon gathering using a balanced kd-tree. We reduced memory usage at the cost of photon tracing time using an Irradiance Cache. We didn't complete everything that we would've liked though. Antiphotons got explored and deemed unnecessary, since photon casting is already fast.

The two authors got together every Tuesday and Friday morning to work on the project together, and we both worked on it on our own. We maintained a Git repository to share code, and kept in communication about the progress and direction of the project. Greg's contribution included shadow photons, printing the results to an image, stabilizing the photon gathering, and triangle meshes. Nate's contribution included balancing the kd-tree, irradiance caching, and parallelization. We collaborated on antiphotons and other miscellaneous work.

Implementation can be challenging. We messed up the .ppm image format before finally getting what should've been easy correct (Figure 8). Shadow photons made some sort of weird caustic at first (Figure 12). Some scenes still have artifacts around the edges (Figure 10). The irradiance cache proved difficult. And ray tracing is still slow with many faces (Figure 13).

5 FUTURE WORK

Future efforts on this project would include ironing out the bugs in some test scenes, by making our photon gathering more robust. We could also implement a k-nearest-neighbors search in the kd-tree instead of the radius query that is currently used, to speed up photon gathering, which is the slowest part of the algorithm so far. And, we could improve the irradiance cache interpolation.

6 SOURCE CODE

We've provided the source code for reference. It's online at https://github.com/ncwheeler700/ACGFinalProject.

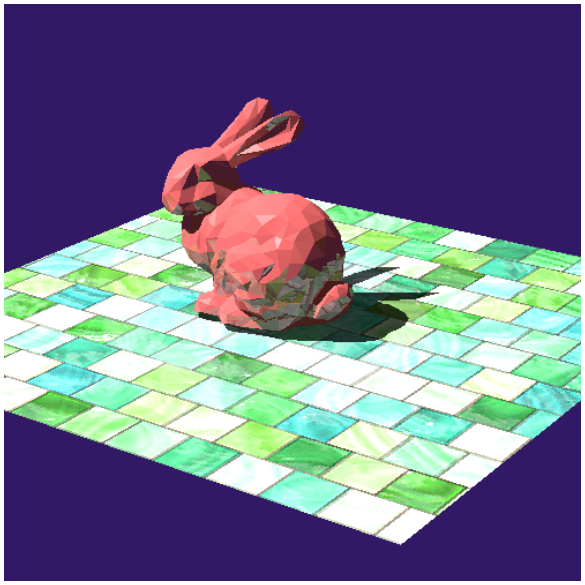


Figure 6: A ray traced red bunny with 1k triangles

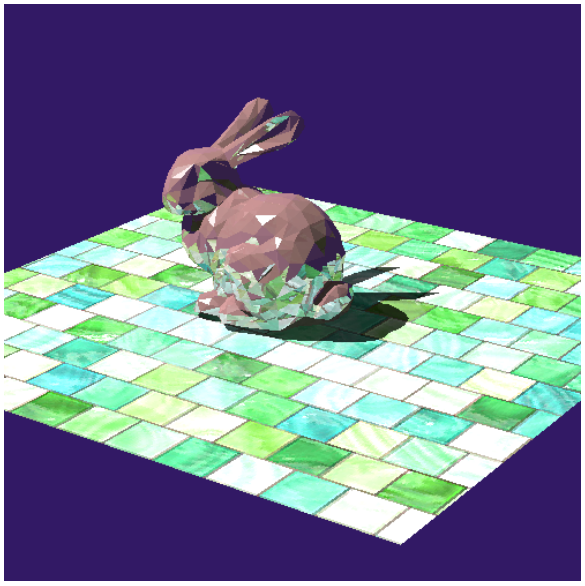


Figure 7: A ray traced red-reflective bunny with 1k triangles

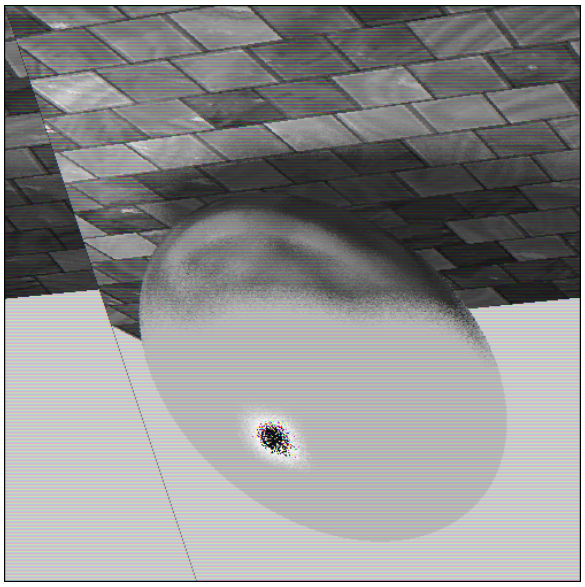


Figure 8: A buggy image from .ppm output

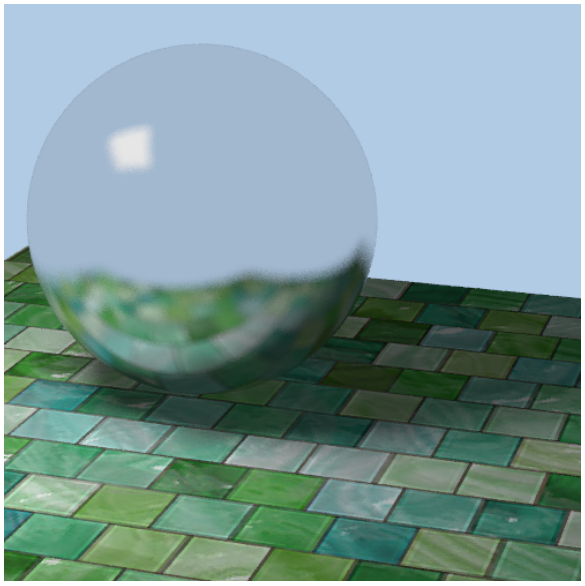


Figure 9: A glossy sphere

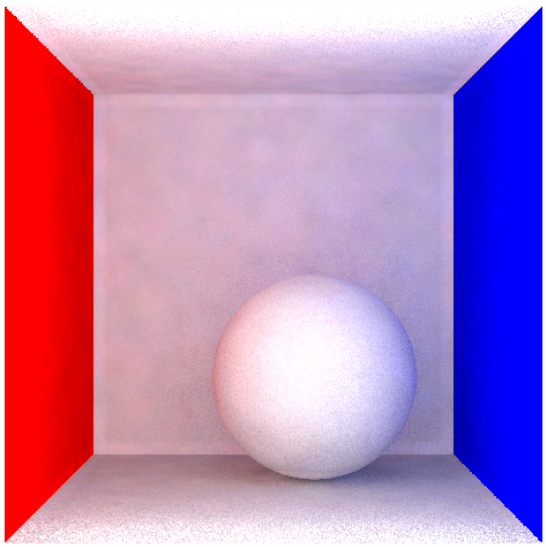


Figure 10: A globally illuminated by photon mapping Cornell Box

ACKNOWLEDGMENTS

The authors would like to thank Professor Barb Cutler for teaching such a challenging and rewarding class. We learned a lot.

REFERENCES

[1] Carsten Dachsbacher, Marc Stamminger, George Drettakis, and Fredo Durand. 2007. Implicit Visibility and Antiradiance for Interactive Global Illumination. In *SIGGRAPH 07 Conference Proceedings*. ACM SIGGRAPH.

[2] Henrik Wann Jensen. 1996. Global Illumination using Photon Maps. In *Rendering Techniques '96*. Eurographics Workshop on Rendering, 21–30.

[3] Henrik Wann Jensen and Niels Jorgen Christensen. 1996. Efficiently Rendering Shadows Using the Photon Map. In *Proceedings of Compugraphics '95*. Eurographics.

[4] Gregory J. Ward, Francis M. Rubinstein, and Robert D. Clear. 1988. A Ray Tracing Solution for Diffuse Interreflection. In *SIGGRAPH '88 Proceedings*. ACM SIGGRAPH, 85–92.

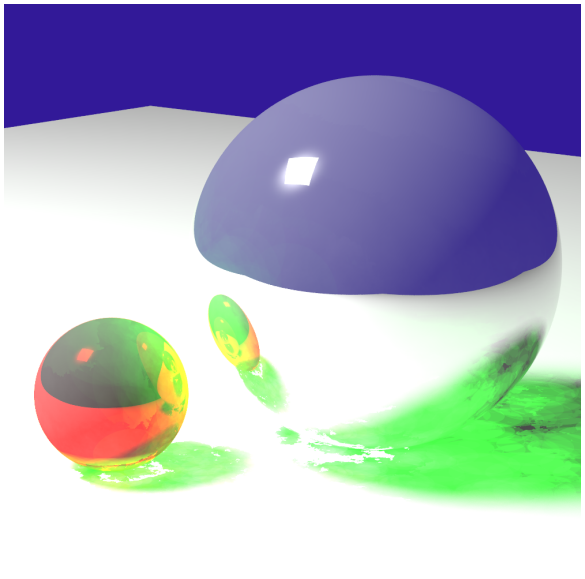


Figure 11: A buggy photon mapping scene

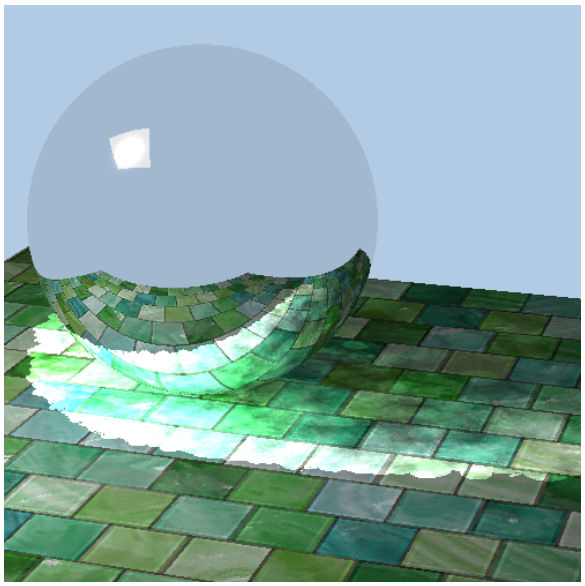


Figure 12: A an early and buggy shadow photons render

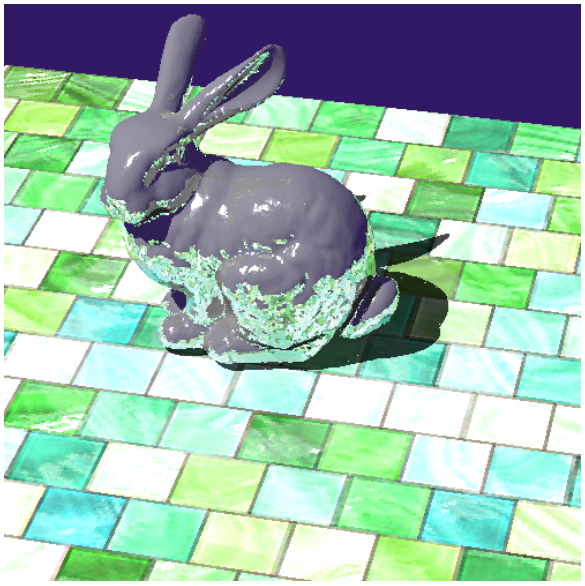


Figure 13: A reflective bunny ray traced with 40k triangles in 105min

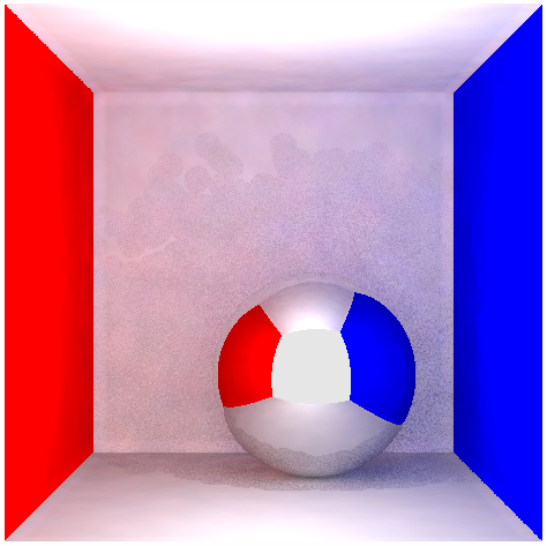


Figure 14: A photon mapped Cornell Box with a reflective sphere