Efficient Implementation of Bi-directional Path Tracer on GPU

Bc. Vilém Otte[*]
Supervised by: RNDr. Marek Vinkler Ph.D.[†]

Faculty of Informatics
Masaryk University
Brno / Czech Republic

Abstract

Most of the implementations solving photo-realistic image rendering use standard unidirectional path tracing, having fast and accurate results for scenes without caustics or hard cases. These hard cases are usually solved by a bidirectional path tracing algorithm. However, due to the complexity of the bidirectional path tracing algorithms, its implementations almost exclusively target sequential CPUs. The following paper proposes a new parallel implementation of the bi-directional path tracing algorithm for the GPUs. Our approach is compared with existing solutions in terms of both performance and image quality. As the references we use the standard unidirectional GPU path tracer and commercial off-line bidirectional path tracers. We achieve interactive rendering rates for scenes of medium complexity.

Keywords: bidirectional path tracing, path tracing, physically based rendering, ray tracing, CUDA

1 Introduction

Global illumination research recently focuses on unbiased methods. Unbiased method is such method, that under some assumptions, converges to the solution of the rendering equation.

One of the common techniques for rendering unbiased images is path tracing, respectively the extension of path tracing - bidirectional path tracing. These techniques are well known for high quality rendering output. Formerly, most of the path tracing and bidirectional path tracing implementations were done on the CPU. However, these techniques are well fit for massively parallel hardware, like the recent generations of graphics hardware.

Even though there are already present implementations of bidirectional path tracing on the graphics hardware, most of the implementations are either using the graphics hardware only for small part of the computation or they do not contain any surface shading at all.

The presented work focuses mainly on the implementation of a full featured bidirectional path tracing renderer, allowing for advanced materials, including textures and computing the entire image using the graphics hardware. The contributions of this paper are the following:

- Design and implementation of three variants of bidirectional pathtracing on the GPU.
- Comparison of their speed and quality to the ground truth path tracing solution.
- Comparison to other GPU based renderers.

2 Related Work

In this section we summarize literature most relevant to bidirectional path tracing - various approaches to global illumination focused on parallel GPU algorithms.

2.1 Ray Casting

The core of almost every global illumination technique is ray casting, which allows for finding closest ray-primitive intersection point along each ray and also to test visibility between two points. This visibility computation is the key to the evaluation of light transport. The focus of research on ray casting are following three issues: selection of acceleration structure, their construction and their traversal.

High performance and well established ray casting solutions are available for public, including NVidia OptiX [11] targeting NVidia hardware (which is actually full ray tracing engine), Intel Embree [15] targeting traditional SIMD-based CPU architectures. Our implementation is based on the open source framework by Aila et al. [1, 2].

2.2 Early Global Illumination

Whitted [14] used ray-casting for generation of photo realistic images, allowing for recursive specular reflections and refractions. Later Cook extended ray tracing to distributed ray tracing to allow for effects such as diffuse interreflection [3]. The rendering equation was introduced in 1986 by Immel et al. [5] and Kajiya [7].

The rendering equation describes light transport in the scene. Light transport describes the energy transfers in a given scene that affects visibility.

[*]vilem.otte@post.cz
[†]xvinkl@fi.muni.cz

2.3 Path Tracing and Bidirectional Path Tracing

Path tracing algorithm allows for computing the solution of the rendering equation. By computing the paths of light from the camera into the scene (eventually reaching light source), it closely resembles the behavior of light. The path starts with primary ray at the camera and is traced into the scene. Upon intersection, in continues in random direction.

For further efficiency, at each vertex of the path it is determined whether path should be terminated or not by Russian roulette, thus preventing infinite path lengths. It allows for physically based computation of lighting, and so the synthesis of photo realistic images. The algorithm is also unbiased, with theoretically absolute accuracy when using infinite number of samples.

Bidirectional path tracing is further extension of path tracing algorithm, introduced by Veach [12]. By generating one sub path from a light source and one sub path from the camera, later joining them together, it is possible to handle indirect lighting computation more robustly (and efficiently) compared to ordinary path tracing. Furthermore, this modification still keeps the resulting algorithm physically based and unbiased.

2.4 Other Global Illumination Methods

Lately a lot of fast algorithms for computation of global illumination were introduced. Most of the algorithms were, compared to path tracing, biased, providing a trade off between rendering quality and speed.

Recent interactive global illumination methods are modifications of Virtual Point Lights, introduced by Keller et al. [9], allowing for real time smooth global illumination [10]

One of the other popular global illumination techniques, used in real time rendering, are Cascaded Light Propagation Volumes [8].

Photon mapping introduced by Jensen [6], is currently popular in production renderers, especially for interior rendering. The technique was further extended into progressive photon mapping [4]. Compared to standard photon mapping, which is a biased rendering algorithm, progressive photon mapping is an unbiased one.

3 Bidirectional Path Tracing

This section presents some improvements when using bidirectional path tracing. For proper description of these features it is critical to define some terms.

Naive Path Tracing This designation is used for path tracers that does not do any explicit steps, but wait for camera ray to actually hit light (or get terminated).

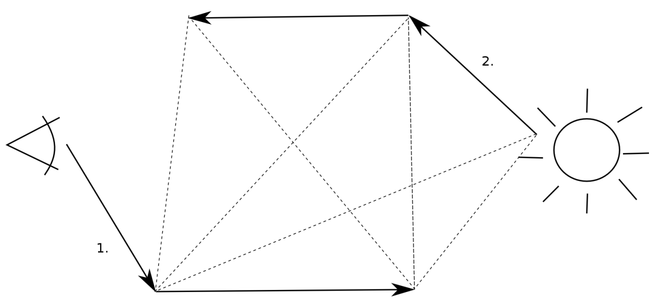


Figure 1: All possible connections between a light path and camera path. 1. Represents camera path, 2. Represents light path, Dashed lines represent possible connections between light path and camera path.

Standard (Explicit) Path Tracing Due to very bad convergence ratio using naive path tracing, it is often understood that generic path tracing algorithm does single explicit step towards light on each vertex of the path. These path tracers are to be designated as standard, or explicit, path tracers.

For the sake of completeness, pseudo algorithms for standard path tracing (see Algorithm 1) and bidirectional path tracing (see Algorithm 2) are provided.

Bidirectional path tracing, computes two paths, one from the light and another one from the camera. These paths are later connected (see Figure 1). These connections attempt to solve several problems introduced by unidirectional path tracers:

Small light sources For naive path tracers, the probability of hitting a light source is proportional to its size. Having point lights (lights that are infinitely small) in naive path tracer often ends up with nothing visible in rendered image, as the probability of hitting the light source reaches zero.

This can be solved by sampling light in explicit manner each step in path computation, resulting in very fast convergence for directly lit scenes. While sometimes this technique is referred as explicit path tracing, it actually is a special case of bidirectional path tracing, where the light path contains only a single vertex.

The visibility function between each camera path vertex and light path vertex has to be computed, resulting in $2 \cdot N$ cast rays for camera path length of N.

For bidirectional path tracing, even more complex paths from the light can be easily evaluated, e.g. light hidden inside a lamp.

Interior scenes lit by exterior light Assuming we are inside a room, where there is only a single window, standard, or even explicit, path tracers are not going to converge very quickly because the probability of sampling the light is low. In fact, in case where no path vertex on camera path lies in direct light, the contribution of that path is

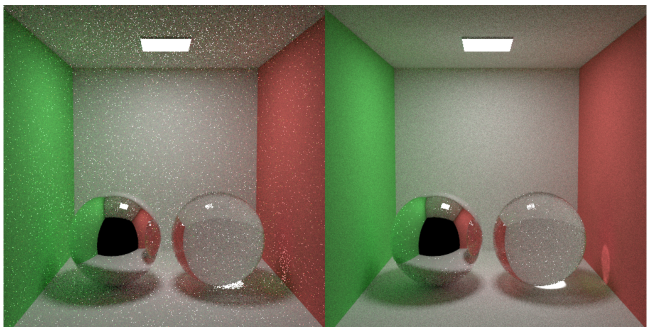


Figure 2: Difference between path tracer and bidirectional path tracer when computing caustics. On the left side, the fireflies generated by standard path tracer compared to more smooth caustics generated by the bidirectional path tracer on the right side.

zero. This is one of the so called 'pathological scenarios' for path tracing, where the algorithm fails to provide result fast enough.

Using a bidirectional path tracer with light path of length M, it is enough that single point of the light path is directly visible from any of the camera path vertices. If this condition is true, that paths' contribution will be non-zero, resulting in faster convergence.

Caustics convergence Using a unidirectional path tracer often results in poor caustics and fireflies in the resulting image, as their convergence rate is by magnitudes worse to convergence rate of diffuse light. Computing caustics with bidirectional path tracer is faster (see Figure 2), yet there are still difficulties with reflected caustics. Bidirectional path tracer can be, however, further extended with Metropolis Light Transport [13], improving the performance on reflected caustics.

Algorithm 1 Path Tracing

```
1: procedure PATHTRACE
2: for each path:
3:     ray ← setup primary ray
4:     while ray.terminated = false do
5:         result ← raycast(ray)
6:         if result.hit = false then
7:             Accumulate background color
8:             ray.terminated ← true
9:         else
10:            Compute and Accumulate surface emission
11:            Compute contribution of random light
12:            if Contribution is non zero then
13:                Accumulate contribution
14:            if Russian roulette terminates path then
15:                ray.terminated ← true
16:            else
17:                ray ← Get B*DF Sample
```

Algorithm 2 Bidirectional Path Tracing

```
1: procedure BIDIRPATHTRACE
2: for each path:
3:     Generate vertex on random light
4:     Push this vertex to light path
5:     ray ← setup light ray
6:     while ray.terminated = false do
7:         result ← raycast(ray)
8:         if result.hit = false then
9:             ray.terminated ← true
10:        else
11:            Push this hitpoint to light path
12:            if Russian roulette terminates path then
13:                ray.terminated ← true
14:    ray ← setup primary ray
15:    while ray.terminated = false do
16:        result ← raycast(ray)
17:        if result.hit = false then
18:            Accumulate background color
19:            ray.terminated ← true
20:        else
21:            Compute contribution by joining light path
22:            with this vertex
23:            if Russian roulette terminates path then
24:                ray.terminated ← true
25:            else
26:                ray ← Get B*DF Sample
```

3.1 Parallel Bidirectional Path Tracing using GPU

Computation of different samples, (sub)paths in terms of path tracing, and different pixels is independent on each other. Path tracing and also bidirectional path tracing are therefore good candidates for massively parallel computation.

The resulting parallel algorithm looks similar to sequential version. The parallel run is performed over the pixels, with each sample is computed by a single thread in a single kernel launch.

4 Implementation

The bidirectional ray tracing kernels were implemented on top of open source framework by Aila. For this purpose, new rendering kernels for each variant of bidirectional path tracer were added.

To achieve high performance of the source code, the speculative while-while traversals were used. As opposing to persistent threads they perform, in general, better on new hardware. Also, bounding volume hierarchy with spatial splits was used as acceleration structure for the rendering.

The Aila framework was further extended to support high resolution textures, reflective, refractive and dielectric materials. Light sources are handled as geometry with emissive material, so in general any number of light sources is supported.

The bidirectional path tracer kernel always processes one pixel in single walk through, e.g. the light path generation (storing required data), followed by computation of the camera path. During each step of the camera path computation the join for currently processed vertex is performed when necessary.

5 Optimizations and Limitations

The following section describes possible optimizations and limitations when implementing bidirectional path tracer on the GPU. For each optimization a brief summary is given.

5.1 Sub path join

Full join

Joining of the light sub path of length M and the camera sub path of length N is a non trivial task. There are multiple ways to join these, the intuitive solution is joining each vertex from light sub path to each vertex in camera sub path. While this actually leads to faster convergence, $N \cdot M$ rays have to be used to compute the visibility between samples, which is slow. Moreover, the full light path has to be stored in the memory, and so larger memory space is required.

Single-step join

Performance wise, the most efficient idea is joining the last camera sub path vertex and the last light sub path vertex. Such approach has some advantages. Single path computation is of the same performance as naive path tracing, although improving the situations where naive path tracing has major problems. However, by combining only the ends of both paths contributions of these light paths are very small.

K-step join

It is also possible to select an approach using randomized algorithm. For each of the $N \cdot M$ joining rays, the algorithm discards some of these joins. The actual joining ray rejection can be built upon multiple criteria - either fully random, or deterministic (removing less contributing joins and accordingly weighting the rest). This join is to be designated as K-step join. The join is performed by taking each vertex from camera against k vertices from the light path. The k value has to be smaller than the number of vertices in the light path.

When implemented properly, the different sub path joins do not break the unbiased property of the algorithm.

5.2 Path pre-generation

Given a static scene, all the light paths can actually be pre-computed. Later, during the execution of the algorithm, we only select one of the light paths from given M pre-computed light paths.

While this approach is highly efficient, it often means that the resulting algorithm is biased. This can be overcome by re-generating these light paths on runtime. Once we start joining samples to random pre-generated paths, it is possible that some samples are to be joined with a single light path. This could lead to unnatural patterns in resulting image, and of course breaking unbiased nature of the algorithm. By re-generating the paths after they have been used, it is possible to avoid this problem.

Unless large M is selected, the quality of resulting image can be highly degraded. To keep the quality of the resulting image high enough, it was experimentally evaluated that the number of light paths must be at least the same as the number of pixels in the resulting image.

5.3 Biasing

Biasing the algorithm does not have much sense for simulations. Although, for performance heavy applications, in case where we have limited time for calculating an image, for example in games, it is possible to trade off quality for speed.

During the implementation, two of the biasing techniques were considered.

Limiting maximum camera path length Generally, the camera path length can be very long (assuming it doesn't directly hit the light), until Russian Roulette finally terminates the path.

By limiting the length of camera path to some value it is possible to increase performance. First of all, longer paths generally tend to have smaller contribution, by ending them at some given maximum length we terminate them early, effectively reducing the number of computations they need to do.

Also, from the GPU perspective, we are always waiting for the longest path to finish the computation, in the worst situation, whole warp is waiting for single thread to finish a very long path. By bounding the maximum length, we effectively reduce this issue and increase the computation performance.

On the other hand, the unbiased nature of the algorithm is lost, by limiting the maximum path length we are effectively limiting the maximum number of reflections/refractions, which may cause visible problems in images (for example, missing reflection).

Random Number Generation As each Monte-Carlo technique, path tracing and bidirectional path tracing, heavily depend on random number generation. Having a random number generator with large period means, that resulting image will converge closely to the ground truth.

Each Monte-Carlo technique spends some amount of time in random number generator. If the application targets performance instead of precision, it is possible to pregenerate random numbers into an array and use those later.

Doing so introduces a limit at which the image is not able to converge more towards the ground truth (as all the samples were already taken).

It is important to note that the results taken using the implementation were recorded with unbiased version of the algorithms. The biased version is between 2 and 3 times faster compared to unbiased version.

6 Results

6.1 Evaluation and Analysis

Results of the implementation were evaluated on low-end laptop GPU NVidia GeForce 720M, with 1.5 GB memory. The system was running under Windows 8.1 OS, with CUDA 5.5 installed. Kernels were compiled with compute capability 2.0. The light paths were generated on the runtime.

A low end GPU was used to demonstrate, that even current generation laptop based GPUs are capable of interactive rendering of moderately complex scenes.

The rendered images were taken each 5 seconds, while the used algorithm was running progressively. Results presented in this section show the difference in quality between the specific implementations. The resulting images are also compared to ground truth using the root mean square error (further RMSE, lower is better).

The first set of comparisons is between path tracing and bidirectional path tracing. Followed by comparison against other GPU based Monte-Carlo rendering systems.

The single-step join does need to keep just a single (last) vertex of the light path. K-step join needs to keep K vertices of the light path in the memory during the computation, while full join needs to keep all the vertices in the light path. While single-step join and K-step join have constant memory footprint, the full join footprint grows with the length of the light path.

6.2 Bidirectional vs. Unidirectional

Cornell Box

We ran two algorithms on the following scene, bidirectional path tracing (with full path join) and standard path tracing. Both of the resulting images are compared to the ground truth. The sample images were taken after 5 seconds and after 10 seconds. (see Table 1)

From the given comparison, it can be stated that bidirectional path tracing with full path joining converges faster compared to standard path tracing. This is especially the case for caustics.

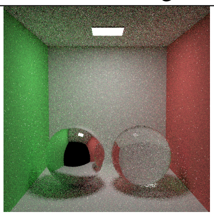
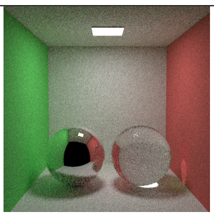
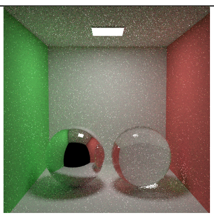
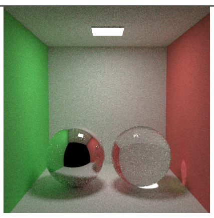
	Path Tracing	Bidirectional
5s		
RMSE	0.1002	0.0628
10s		
RMSE	0.0784	0.0456

Table 1: comparison between Path Tracing and Bidirectional Path Tracing with full path join.

Crytek Sponza

The following scene shows complex materials, like textured, alpha-tested, reflective and refractive surfaces, lit by an area light source.

All three different bidirectional kernels were run on this scene, and the results show how the scene looked after 5 seconds of processing. All the results are compared to the ground truth in terms of RMSE (see Table 2).

We observed the same behavior in all of our measurements, full path join results in best image quality and fastest speed. This shows, that the algorithm is not bound by memory performance, in which case K-step join or single step join would result in higher performance and thus quality on a fixed budget.

Caustics Test

We also ran three different bidirectional kernels, as well as standard path tracing method, on a scene with a caustic. The results show how the scene looked after 5 and 10 seconds of processing (see Table 3).

The full path join also results in the highest quality caustics, that are almost perfectly smooth after ten seconds of computation.

6.3 Other GPU rendering packages

iRay

NVidia iRay is a physically based renderer highly scalable in performance across GPUs and CPUs. We rendered the Crytek Sponza scene using our bidirectional path tracing renderer and the iRay from a similar viewpoint (see Table 4).

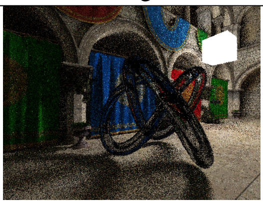
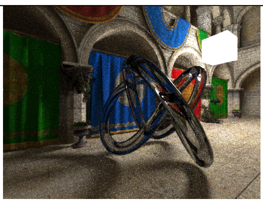
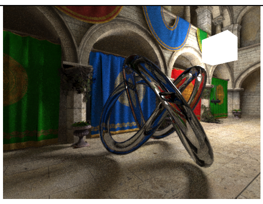
Renderer	Image	RMSE
Single-step		0.1584
K-step (K=3)		0.0701
Full		0.0095

Table 2: Equal-time comparison between three different implementations of bidirectional path tracing and their distance to ground truth in terms of RMSE.

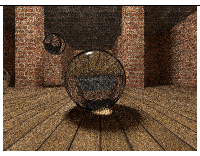
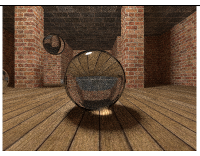
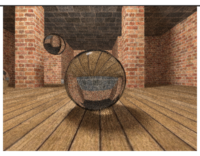
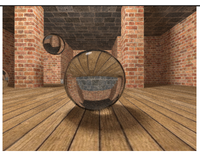
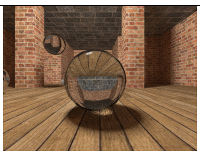
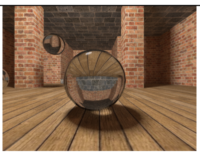
Renderer	5s	10s
Single-step		
K-step (K=3)		
Full		

Table 3: Comparison of a caustic scene with three different implementations of bidirectional path tracing. The contrast was intentionally enhanced so it is possible to see the difference in sampling inside caustics casted by a glass sphere. Single step join has very slow convergence and so the resulting image is darker compared to the others.

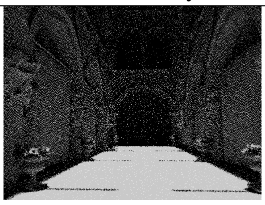
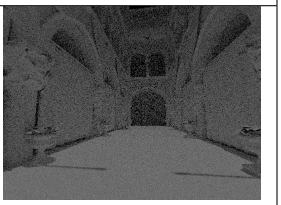
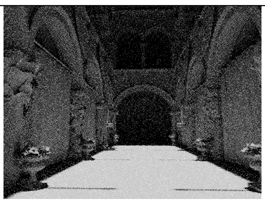
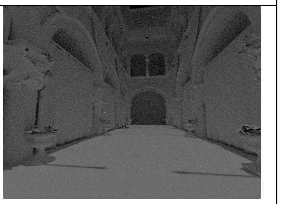
	NVidia iRay	Bidirectional
5s		
10s		

Table 4: Comparison between iRay and our bidirectional path tracing implementation. The brightness/contrast difference is caused by different handling of output between both implementations.

Again we target images generated after 5 and 10 seconds, which shows the quality achievable using an interactive preview on low end graphics card.

The resulting images are untextured, as there is not a full support for textured surfaces in iRay, and the scene was lit using single directional light. This setting also allows for an easier comparison of the quality of the global illumination without the masking effect of the textures.

The technique iRay uses is actually standard path tracing, in comparison it is clearly visible that our Bidirectional technique produces smoother results for a similar time budget.

LuxRender

LuxRender is a physically based and unbiased rendering engine. The LuxRender package was used through Blender software, possibly limiting some options leading to slightly decreased quality (see Table 5).

LuxRender uses a technique called LuxRays to produce the image, it is an unbiased technique similar to bidirectional path tracing. The implemented bidirectional solution might seem to converge better, although it is important to state, that LuxRender is a very large package supporting complex material setup (along with subsurface scattering effects for example), while the implemented solution does not.

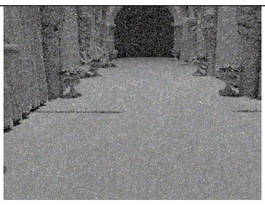
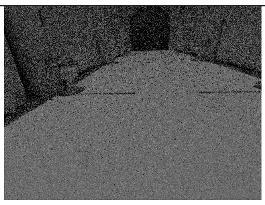
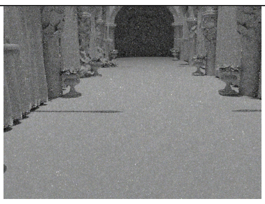
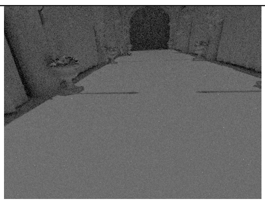
	LuxRender	Bidirectional
10s		
20s		

Table 5: Comparison between LuxRenderer and our bidirectional path tracing implementation. The brightness/contrast difference is caused by different handling of output between both implementations.

7 Summary

Path tracing, in general, excels in exterior scenes or when there are large light sources. Bidirectional path tracing is an extension that allows for faster computation of more complex situations, like interior scenes or small light sources.

The created GPU-based implementation showed, that bidirectional path tracing is also suitable for massively parallel implementation. The results confirmed, that convergence rates are in general better for bidirectional path tracer, which effectively reduces computation time.

Full path joining proved to lead to the highest quality global illumination, although keeping large memory footprint. However, on modern GPU architectures, the memory footprint required by full path joining is bearable.

K-path join proved to be an interesting alternative to full path join. Even though the full path join actually results in better convergence rates, it is possible to alter the number of joins for K-join on the fly. This can be used to achieve a constant refresh rate when running the algorithm in progressive mode. Such approach can be interesting for interactive preview of rendered scene.

References

[1] Timo Aila and Samuli Laine. Understanding the efficiency of ray traversal on gpus. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, pages 145–149, New York, NY, USA, 2009. ACM.

[2] Timo Aila, Samuli Laine, and Tero Karras. Understanding the efficiency of ray traversal on GPUs – Kepler and Fermi addendum. NVIDIA Technical Report NVR-2012-02, NVIDIA Corporation, June 2012.

[3] Robert L. Cook, Thomas Porter, and Loren Carpenter. Distributed ray tracing. In *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '84, pages 137–145, New York, NY, USA, 1984. ACM.

[4] Toshiya Hachisuka, Shinji Ogaki, and Henrik Wann Jensen. Progressive photon mapping. *ACM Trans. Graph.*, 27(5):130:1–130:8, December 2008.

[5] David S. Immel, Michael F. Cohen, and Donald P. Greenberg. A radiosity method for non-diffuse environments. *SIGGRAPH Comput. Graph.*, 20(4):133–142, August 1986.

[6] Henrik Wann Jensen. Global illumination using photon maps. In *Proceedings of the Eurographics Workshop on Rendering Techniques '96*, pages 21–30, London, UK, UK, 1996. Springer-Verlag.

[7] James T. Kajiya. The rendering equation. In *Computer Graphics*, pages 143–150, 1986.

[8] Anton Kaplanyan and Carsten Dachsbacher. Cascaded light propagation volumes for real-time indirect illumination. In *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '10, pages 99–107, New York, NY, USA, 2010. ACM.

[9] Alexander Keller. Instant radiosity. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '97, pages 49–56, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[10] Jaroslav Křivánek, Iliyan Georgiev, Toshiya Hachisuka, Petr Vévoda, Martin Šik, Derek Nowrouzezahrai, and Wojciech Jarosz. Unifying points, beams, and paths in volumetric light transport simulation. *ACM Trans. Graph.*, 33(4):103:1–103:13, July 2014.

[11] Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. Optix: A general purpose ray tracing engine. *ACM Trans. Graph.*, 29(4):66:1–66:13, July 2010.

[12] Eric Veach. *Robust Monte Carlo Methods for Light Transport Simulation*. PhD thesis, Stanford, CA, USA, 1998. AAI9837162.

[13] Eric Veach and Leonidas J. Guibas. Metropolis light transport. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '97, pages 65–76, New York,

NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[14] Turner Whitted. An improved illumination model for shaded display. *Commun. ACM*, 23(6):343–349, June 1980.

[15] Sven Woop, Louis Feng, Ingo Wald, and Carsten Benthin. Embree ray tracing kernels for cpus and the xeon phi architecture. In *ACM SIGGRAPH 2013 Talks*, SIGGRAPH '13, pages 44:1–44:1, New York, NY, USA, 2013. ACM.