Fast Photon Gathering in Progressive Photon Mapping Using GPGPU

Tomaáš Lysek*
Supervised by: Pavel Zemčík†

Department of Computer Graphics and Multimedia
Faculty of Information Technology, Brno University of Technology
Brno / Czech Republic

Abstract

This paper describe an experimental method implementing Progressive photon mapping on contemporary hardware - PC with GPU. The overview of Progressive photon mapping as well as its GPGPU-specific modification. The experimental results are shown as well along with the performance measurements.

Keywords: Global illumination, photon mapping, gpgpu, opencl

1 Introduction

From the beginning of computer graphics, there was an effort to make nice photorealistic images. One way to do this is to use global illumination methods. These methods have high computational cost and they are not always suitable for massive parallelism on graphics acceleration hardware, such as GPU. Photon mapping is a relatively new approximation of global illumination and Progressive photon mapping is its very promising variant. This paper propose a naive implementation of Progressive photon mapping on contemporary GPU and propose acceleration approaches on this naive solution.

2 Related work

Best photorealistic rendering results today are achieved using methods belonging to the global illumination ones. These methods attempt to simulate physically correct light propagation in scenes, and basis for all of the global illumination methods was set in 1986 when James T. Kajiya [5] formulated the rendering equation.

Rendering equation describes how to precisely compute reflected radiance in a certain point by summing light contribution from all direction in hemisphere around examined point. Using this approach, it is possible to compute precise light propagation in the scene. The major problem of this approach is that this computation is done by computing integral through hemisphere and computing this integral is nearly computationally impossible for large scenes.

In the paper that introduced the rendering equation, the path tracing method was proposed as well. Path tracing samples Rendering equation by examination random direction in hemispheres. To get good result, very many random paths have to be computed. Path tracing is, in fact, using Monte Carlo approach for integral computation and thus the methods based on path tracing are called Monte Carlo methods.

Path tracing was probably the first complete global illumination technique (although radiosity was invented earlier, it assumes only diffuse light propagation) and using this technique, it was possible to compute nice photorealistic images. Extension of path tracing called bidirectional path tracing was invented independently in 1993 [6] and 1994 [7]. Bidirectional path tracing traces paths from eye and from light simultaneously and from this two paths, it computes illumination. It is possible (in scenes with lot of indirect illumination) to compute photorealistic images in lower time using bidirectional path tracing comparing to the original path tracing. Another extension of path tracing is so called Metropolis light transport [8] and this technique sets another examinated path from previous path by mutation of such path.

To achieve good photorealistic results using Monte Carlo raytracing methods, many paths per pixels need to be examined and it is very time consuming. Another way how to approximate rendering equation is using Photon mapping. Photon mapping was invented by Henrik Wann Jensen in 1996 [4]. It is two-pass algorithm; in the first pass light contribution in scene is computed by sampling light distribution from scene, sample of light is called photon and the photons are saved in photon maps covering surfaces of the scene. In the second pass, an extended raytracing is used to compute final image. When this extended raytracing computes local illumination model (for example by phong lightning), illumination from photon maps is included.

For good results in Photon mapping, large number of

*xlysek03@stud.fit.vutbr.cz
†zemcik@fit.vutbr.cz

photons has to be saved in photon map and if photon map is really big, searching in photon map becomes slow. Extension of normal photon mapping, called Progressive photon mapping [2] addresses this problem. The difference from the standard Photon mapping is that Progressive photon mapping computes many smaller photon maps and progressively improves results from another photon maps. In 2009, Stochastic progressive [1] photon mapping was proposed, extending Progressive photon mapping by distributed raytracing effects, such as depth of field, motion blur, etc.

All of the above described global illumination methods are consistent - this means that with increasing rendering time method is approaching correct result. Monte Carlo raytracing methods are unbiased - this means that even if we compute one path per pixel and average large amount of this images, we still get correct result. On the other hand Photon mapping methods are biased and if we average many of rendered images we do not generally get correct result.

3 Progressive photon mapping

Radiance estimation in photon mapping is an approximation of Rendering equation. Luminance at point x heading in direction $\vec{\omega}$ is computed as:

$$L_r(x,\vec{\omega}) \approx \frac{1}{\pi r^2}\sum_{p=1}^{N} f(x,\vec{\omega},\vec{\omega_p})\Delta\Phi_p(x,\vec{\omega_p}) \qquad (1)$$

where $\Delta\Phi_p(x,\vec{\omega_p})$ is photons flux saved in photon map. $f(x,\vec{\omega},\vec{\omega_p})$ is bidirectional reflectance distribution function. Sum involves N nearest photons from point x in photon map. Nearest photon in space generates sphere and because it is possible to assume that photons are accumulated from flat surface, the result is divided by area of circle where r is distance of farthest photons from point x.

For removing low-frequency noise in result images, photon map has to have many photons, possibly infinitely many. If photon map has infinite number of photons and in radiance estimation is gets fraction of this infinite photons, radiance is estimated in radius approaching zero at the limit. From this observation, it is possible to assume that best results are obtained by radius as small as possible. With increasing number of photons in the photon map, both memory and time complexities are increasing.

There was an effort to divide final large photon map into several smallest photon maps, compute some sort of data and get better result in faster time. One way was averaging lot of computed images, but this does not lead to consistent result.

Another way was invented with Progressive photon mapping. This multiple-pass technique reorders photon mapping in proper way and ensures that with another passes, consistent result is obtained.

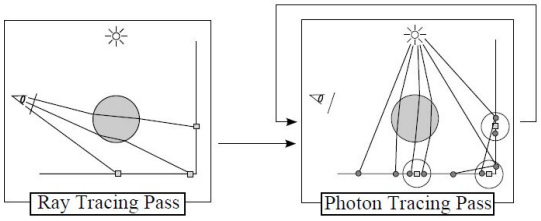

Figure 1: Progressive photon mapping schema[2]

Scheme of Progressive photon mapping is shown in Figure 1. First, raytracing is performed and after this, photon tracing passes are performed. Raytracing is performed for saving special positions in scene - so called hitpoints. Hitpoints are saved when ray intersect with diffuse surface, and in each hitpoint lot of needed data are saved. This data contains: position, material, normal, pixel location, pixel weight, current photon radius, accumulated photon count and accumulated reflected flux. Using this data it is possible to synthesize final image.

Photon tracing pass is divided into iterations, with fixed (smaller than in normal photon mapping) number of photons. Theoretically it is possible to process infinite number of photons on limited memory. In each iteration, new random photon map is created and then for each hitpoint, illumination from photon map is computed. As it was described above, radius decreases with each iteration. Equation for computation new radius is written as:

$$\hat{R}(x) = R(x)\sqrt{\frac{N(x)+\alpha M(x)}{N(x)+M(x)}} \qquad (2)$$

where $R(x)$ respectively $\hat{R}(x)$ is radius in hitpoint x respectively new radius in hitpoint x. $N(x)$ is number of photons saved in hitpoint x, $M(x)$ is number of new photons in current iteration in current radius $R(x)$. α is value in range 0 - 1 and indicate how much new photons will be added to illumination and how fast radius will be decreasing.

New flux $\tau_{\hat{N}}(x,\vec{\omega})$ in hitpoint x is computed as:

$$\tau_{\hat{N}}(x,\vec{\omega}) = \tau_{N+M}(x,\vec{\omega})\frac{N(x)+\alpha M(x)}{N(x)+M(x)} \qquad (3)$$

where $\tau_{N+M}(x,\vec{\omega})$ is sum of flux from previous iteration and current iteration computed by equation 1 and values $N(x), M(x)$ and α is same values as in equation 2.

Final luminance in point x heading in direction $\vec{\omega}$ is computed as:

$$L(x,\vec{\omega}) \approx \frac{1}{\pi\hat{R}(x)^2}\frac{\tau_{\hat{N}}(x,\vec{\omega})}{N_{emmited}} \qquad (4)$$

It is possible to render image after each photon tracing iteration or after all iteration has been proceed.

Using progressive photon mapping it is possible to render whole global illumination and very similar images like using monte carlo raytracing methods. Progressive photon mapping excel in rendering specular diffuse specular path (SDS path) above all other monte carlo raytracing methods [2]. SDS path means that path from light is connected through any number of specular surfaces, reflected on diffuse surface and then again reflected through any number of specular surfaces to eye. This effect can be seen for example on bottom of swimming pool.

4 GPGPU model

Graphics Processing Units have a unique ability to accelerate general purpose tasks. Graphics cards were made in order to accelerate realtime computer graphics; however, lately the GPUs have been able to compute nearly any kind of programmable task.

Few programing languages exist for programing GPUs, the most used being CUDA and OpenCL. They vary by notation and capabilities and because this paper is using OpenCL for execution, OpenCL notions will be presented.

Execution model

Graphics card consist of a set of streaming multiprocesors. Streaming multiprocessors can be viewed as big enhanced SIMD (single instruction single data) processors. One thread of parallel computation is called kernel. Kernel are grouped into work group - it is set of kernels executed on same streaming multiprocessor. When computation is set on gpu, size of workgroup has to be set and size of final number of threads have to be set.

Memory model

GPU executes many threads simultaneously, so it is not possible to allow all operate with memory. In GPU, three memory space exist. The first is a large global memory. This memory is the biggest memory on GPU (in size of gigabytes) and has the slowest access time of all memory on GPU.

Other memory is the local memory available only to threads in one workgroup; this memory is called local memory and size of this memory is in the order of tens of kilobytes. THe access time of local memory is much faster than to the global memory because local memory is on the same chip as the streaming multiprocessors while the global memory is on another chip (because of it's bigger size).

The third memory space is private memory space and in this memory is exclusive only to one thread. This memory is also mapped on registers and is used for variables, counters, etc. This memory has fastest access time but smallest size.

GPU architecture is very different compared to the CPU. The program execution must satisfy some requirements for fast execution. Memory operation has to be coherent or has to be in block. This means that all threads in one workgroup has to read from one memory position or from block of memory. Access time of memory operation is much bigger than on CPU. From this requirements it is clear that for programs with lot of memory operation gpu is not beneficial as for program with less memory operations.

5 Simple GPGPU decomposition

Progressive photon mapping could be divided into several blocks: raytracing, photon tracing, hitpoint illumination and image synthesis.

Raytracing

Raytracing could be understood as sequential examining each pixel's color in final image. Examination routine of one ray is same for each pixel, so it is possible to implement to one kernel examination of one ray. Global work size of raytracing task is equal to number of pixels in final image rounded to size of work group.

Raytracing routine is often written in recursive manner on CPU. This recursive approach is not possible to use on current GPGPU, because GPGPU programming languages do not allow recursive function calling. Therefore, raytracing has to be written without recursion, stack or dynamic array. One possible way of doing this is make iterative function calling (with maximum iteration) and called function will return another ray.

Most scenes are described by set of triangles. There are exists lot of ray-triangle intersection algorithms, chosen technique in own implementation of simple GPU raytracing is Havel's algorithm [3]. When ray is examining with scene, it has to be tested through all triangles in scene.

Solving this problems and combining them into one kernel it is possible to get naive GPGPU implementation of raytracing.

Photon tracing

Photon tracing block uses very similar routines for scene traversal like raytracing. Each initial photon is traversed in separate kernel. The problem occurs when photon has to determine random direction, in photon generating or in photon reflection on diffuse surface. GPGPU does not have any sort of random generator and therefore random generator is needed. It is possible to use classic congruential generator. Same as in raytracing, photon tracing has set max recursion value. This is done, because before starting photon tracing kernel memory to fixed size have to be allocated.

Hitpoint illumination

After each photon tracing pass, new illumination for each hitpoint has to be calculated. It is only computation few formulas if we get all information needed. The slowest thing in whole hitpoint illumination is finding for each hitpoint x photons in radius $R(x)$.

Very naive solution is going through all photons, compute distance from each photons and process only those photons with radius lower than $R(x)$.

Image synthesize

Image synthesize is possible to perform after each photon tracing and hitpoint illumination pass or at the end of all iterations. All needed data are saved along with each hitpoint and therefore computation of luminance is very easy. For each hitpoint will be executed one kernel and this kernel will compute color for his hitpoint. This color will scale by hitpoint weight and atomically add to framebuffer in proper position given by hitpoint framebuffer position.

6 Evaluation of simple GPGPU implementation

Naive implementation was done for evaluating bottlenecks of progressive photon mapping on GPGPU. This naive implementation was done on very simple scene only for experimental usage and for evaluating biggest bottleneck in whole process. Image 2 show results of this simple implementation. Implementation of progressive photon mapping was done on GPGPU and on CPU to evaluate performace between this two execution possibilities.

CPU implementation was evaluated on laptop with Intel i7-4702MQ processor written in c++ and was compile with Intel c++ 15.0 compiler. GPGPU implementation was written in OpenCL and was evaluated on nVidia GeForce GT 750M. Speed of CPU implementation was evaluated using std::chrono library and GPU implementation was evaluated using nvidia nsight timeline profiler.

	GPU	CPU
Raytracing	0.706 ms	1172 ms
Photon tracing	7.897 ms	317 ms
Hitpoint Illumination	2041 ms	1593 ms
Synthesize	0.247 ms	3 ms

Table 1: Performance evaluation between CPU and GPU naive implementation with 320*280 resolution and 100 thousands photons in photon tracing pass

Table 1 and table 2 show performance between GPU and CPU implementation for 320*280 and 1920*1080 resolution. In each photon tracing iteration 100 thousands photons was traced. As it can be seen, GPU is far more

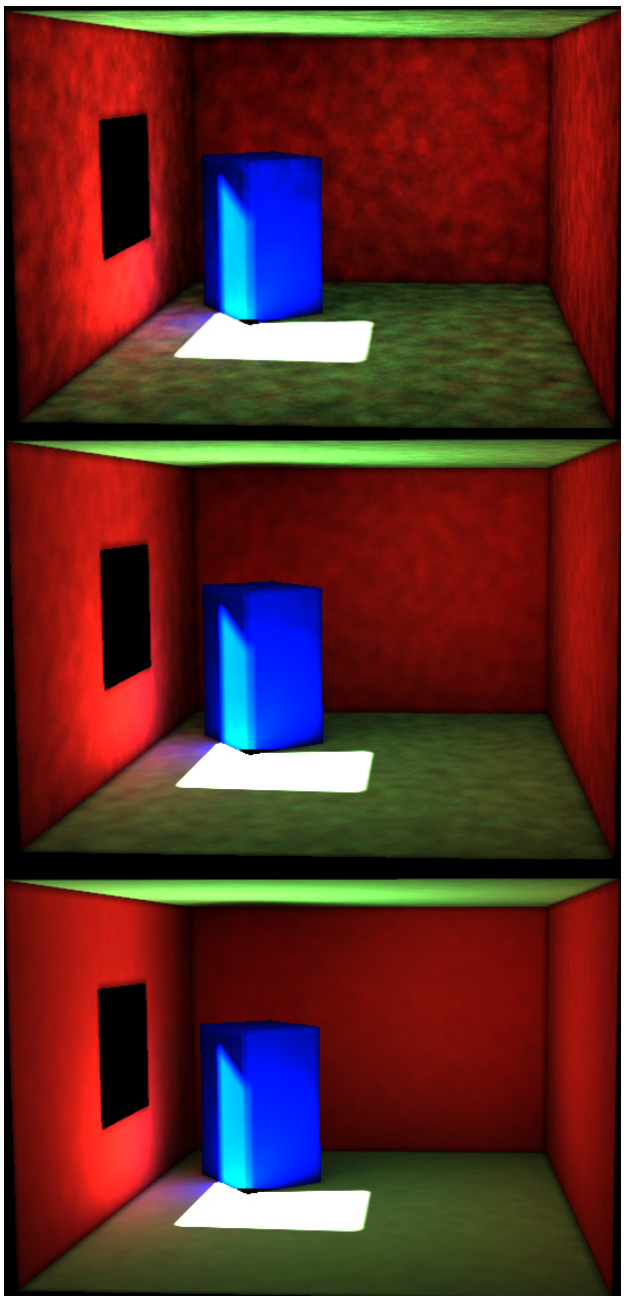

Figure 2: Progressive results of photon mapping. On top image is result with one iteration, on the middle image is result with 10 iterations, on the bottom image is result with 100 iterations. In each iteration 100 thousands photons was generated

faster than CPU implementation in all blocks except hitpoint illumination. CPU block of hitpoint illumination is using kd-tree and if this block will use bruteforce search similar to GPU it will be much more slower. For resolution 320*280 bruteforce on CPU was performed in 68 seconds.

Speed of raytracing and photon tracing is influenced by number of pixels / photons and by complexity of scene. Even when testing scene was simple, photon tracing and

	GPU	CPU
Raytracing	12.107 ms	6 197 ms
Photon tracing	7.897 ms	317 ms
Hitpoint Illumination	38 603 ms	23 957 ms
Synthesize	4.924 ms	91 ms

Table 2: Performance evaluation between CPU and GPU naive implementation with 1920*1080 resolution and 100 thousands photons in photon tracing pass

raytracing aren't bottlenecks because lot of accelerating algorithm for spatial subdivision exists and it is possible to use existing solution for this task for example nvidia OptiX framework. Because of this, this paper will not focus to implementing fast raytracing.

Hitpoint illumination is slowest block in progressive photon mapping implementation on GPU and on CPU. This fact is influenced by very naive implementation of hitpoint illumination and this paper will furthermore focus on accelerating hitpoint illumination on GPGPU.

7 Accelerated photon gathering

Bruteforce implementation of hitpoint illumination is very naive and very slow. In this simple implementation, each thread in workgroup load one photon from memory, compute distance, check if distance is lower than proper radius and optionally accumulate photon flux.

Memory loads are very costly on GPU, even if streaming multiprocessor is multiplexing work between few workgroups, it will wait long time in sleep mode to load data from global memory. One way how to mitigate memory waiting is to use local memory.

This section will present accelerating approaches to accelerate this task. Mostly every other approach will be based on previous approach.

Local memory

Idea is to load block of photons into local memory and go through this photons from local memory. Access to local memory is faster than access to global memory, so it should lead to acceleration.

Each thread in workgroup will load one photon into local memory. Kernel driver will recognize this as block loading and this block loading should be done faster than sequential load of each photon.

Photon sorting

When illumination is computed for hitpoint x, in computation have to be included only those photons lying on same mesh as hitpoint x. When illumination is computed by bruteforce, in hitpoint illumination photons from all mesh are included and when photon from other mesh is loaded to illumination computation, this photon are discarded and costly load was in vain.

One way, how to solve problem with loading photons from different meshes, is to sort photons into unique photon map for each mesh. This task should be done in separate kernel. Each thread will load one photon, check its mesh, by special operation called atomic inc will get position in separated photon map and this photon will save into proper position in memory.

When hitpoint is illuminated it will check hitpoint mesh look into proper photon map and will sequentially loading photons from proper photon map. In this case it is not possible to use local memory because it is not guaranteed that all hitpoints in workgroup will lie on same mesh.

Hitpoint sorting

It is needed to ensure that all hitpoints in workgroup lie on same mesh to use coherent approach (all thread are reading from same memory location) to fast loading from memory.

Hitpoint sorting could be done on CPU size because this work will done only once per whole render process. Hitpoints are loaded to CPU side after raytracing pass. This hitpoints are sorted to separate arrays. In the end of each hitpoint array, special filler hitpoints have to be saved.

This is because ending workgroup of one hitpoint array is not filled fully, for instance we have size of workgroup 256 and in last workgroup of hitpoint array is only 150 hitpoints, so if we do not save 106 filler hitpoints, this last workgroup will consist from hitpoints from two meshes.

If it is ensured that hitpoints in all workgroup is same, performance should accelerate because all threads in workgroup is loading sequentially from same memory.

Hitpoint sorting and local memory

If all photons are separated and similar hitpoints are saved in one workgroup it is possible to use local memory to save photons from separated photon map. This should lead to best acceleration and should be much faster than naive sequential loading.

Hitpoint clustering and spatial grid

Last proposed acceleration is to build spatial grid on each scene objects and sort photons to appropriate subspaces. Then all hitpoints will search only through limited space and will not go through all photons on one mesh. For speedup purposes photons in each workgroup photons should be as close as possible, so some sort of clustering is needed.

8 Experimental results

Acceleration approaches introduced in previous section was implemented on same scene. Table 3 shows result

of this implementation.

	320*280	1920*1080
Naive solution	2041 ms	38 603 ms
Local memory	1109 ms	17 750 ms
Photon sorting	611 ms	10 539 ms
Hitpoint sorting	484 ms	8 384 ms
Hitpoint sorting, local memory	327 ms	5 422 ms

Table 3: Performance evaluation between all proposed acceleration techniques.

As it can be seen, all proposed approaches lead to some sort of speedup. Local memory accelerate computation nearly twice. Photon sorting accelerate naive solution nearly four times. Using this approach special kernel execution is needed. Kernel for photon sorting has been executed in nearly 1ms, so final speedup of hitpoint illumination is much higher than overhead made by special kernel execution.

Hitpoint sorting uses photon sorting with sorted hitpoints on CPU side, so some CPU overhead is needed. Because this task is performed only once - after raytracing pass - this overhead does not matter. Last accelerating approach use local memory with hitpoint sorting and photon sorting. This approach is much faster then all previous approaches.

All of this test was written in OpenCL. OpenCL has built-in function distance() and even when this function should be very fast, it isn't. When on last, fastest approach was distance() function replaced by manual distance calculation from high school, this block get nearly three times speedup! Hitpoint illumination with hitpoint sorting and local memory extended by manual distance calculation has 125 ms execution on 320*280 resolution and 1 941 ms execution on 1920*1080 resolution.

Hitpoint clustering and spatial grid

For clustering, k-means algorithm was used. K-means algorithm return hitpoints clustered in close clusters. One problem occurs with this type of clusters, it is not possible to made fix number of hitpoints in one cluster so lot of filler hitpoints have to be used to fill gaps in hitpoints so one workgroup will only consist from hitpoints from one cluster and eventually filler hitpoints. Lot of filler hitpoints have to be used with k-means. For 1920*1080 resolution nearly 33% filler hitpoints (in sum of all hitpoints) was generated and this lead to slower performance without using grid.

Image 3 shows how clusters are made in scene. For each object in scene, simple (naive) grid structure is proposed. First bounding box of object is computed, then longest axis is split by fixate number of pieces. Length of one divided piece give length of cube of one subspace and then this subspaces are uniformly distributed on object. For each subspace on each object maximum size (in photons count) have to be set. This approach lead to easy implementation and this approach has nice modification options.

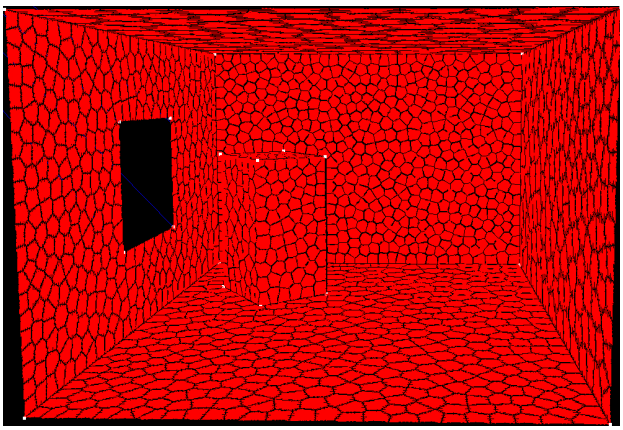

Figure 3: Hitpoint clusters made by k-means

	320*280	1920*1080
Hitpoint sorting, local memory	172 ms	2 852 ms
Grid structure	87 ms	730 ms

Table 4: Performance evaluation of naive grid structure with clustered hitpoints.

Table 4 shows that by using clustered hitpoints and naive spatial grid it is possible to achieve double acceleration.

9 Conclusion

This paper described an experimental implementation of Progressive photon mapping by its decomposition into several blocks suitable for GPGPU. More or less naive implementation of these blocks in GPGPU was proposed as well. The full Progressive photon mapping performance in GPGPU was tested and compared to the CPU. The slowest block turned out to be Hitpoint illumination and for this block, an acceleration approaches were proposed as well.

The Fastest evaluated approach was sorting photons into a mesh, sorting hitpoints into a mesh as well, compute clusters on each mesh and use grid for photon acceleration search. Used grid is very naive and in future work this grid should be replaced by more complex and efficient grid, but this very naive grid with clustered hitpoints shows way to accelerate hitpoint illumination block. Another thing in future work should aim to efficient clustering with low filler hitpoint ratio.

The future work also includes the overall profiling of the GPGPU Progressive photon mapping implementation and also various planned improvements in quality of rendering and speed.

References

[1] Toshiya Hachisuka and Henrik Wann Jensen. Stochastic progressive photon mapping. *ACM Trans. Graph.*, 28(5):141:1–141:8, December 2009.

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

[3] Jiri Havel and Adam Herout. Yet faster ray-triangle intersection (using sse4). *IEEE Transactions on Visualization and Computer Graphics*, 2010(3):434–438, 2010.

[4] 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.

[5] James T. Kajiya. The rendering equation. *SIGGRAPH Comput. Graph.*, 20(4):143–150, August 1986.

[6] Eric P. Lafortune and Yves D. Willems. Bi-directional path tracing. 1993.

[7] Eric Veach and Leonidas Guibas. Bidirectional Estimators for Light Transport. 1994.

[8] 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.