B. Comp. Dissertation

Real Time Path Tracing on GPU

By

Lin Daqi

Department of Computer Science

School of Computing

National University of Singapore

2016/2017

Project No: H113540

Advisor: Dr. Kok-Lim LOW

Deliverables:

Report: 1 Volume

Abstract

With the rapid development of general purposed computing power on graphics hardware, path tracing has been an increasingly popular topic for research for its photorealistic effect based on straightforward concept, wide application in film and gaming industry and huge potential for adaptation in real-time rendering in the future. However, most of today's mainstream path tracing software only supports off-line image synthesis without the capability of real-time interaction and manipulation. To demonstrate the potential of real-time rendering, I will introduce a light-weighted, highly efficient path tracer built with CUDA architecture which encompasses almost all essential functionality of mainstream path tracers, while allowing real-time user interaction with camera and scene geometry. In addition, I will introduce some existing and new algorithms used in my path tracer which significantly improves tracing speed and convergence rate in current GPGPU.

Subject Descriptors:

I.3.5 Computational Geometry and Object Modeling

I.3.7 Three-Dimensional Graphics and Realism

Keywords:

Real-time Path tracing, GPGPU, spatial acceleration structure, Monte Carlo algorithm, BSDF, GPU SAH kd-tree construction

Implementation Software and Hardware:

CUDA 8.0, Visual Studio 2015, Blender, NVIDIA GeForce GTX 960M

Acknowledgment

Highest appreciation for my supervisor, Dr. Kok-Lim LOW’s invaluable guidance.

Table of contents

Title		i
Abstract		ii
Acknowledgement		iii
1	Introduction	1
	1.1 Background Information	1
	1.2 Project Objectives	2
	1.3 Layout of Thesis	2
2	Overview of Software Workflow	4
3	Spatial Acceleration Structure	7
	3.1 Choice of SAS	7
	3.2 Surface Area Heuristics	7
	3.3 Triangle Clipping in Kd-tree Construction	8
	3.4 Kd-tree Traversal	10
	3.5 SAH-based BVH	12
	3.6 Automatic BVH Refitting	13
4	Sampling Algorithms	14
	4.1 The Rendering Equation	14
	4.2 Stratified Sampling vs. Anti-aliasing Filters	15
	4.3 BSDF and Importance Sampling	16
	4.4 Next Event Estimation	17
	4.5 Fresnel Switch and The Russian Roulette	19
	4.6 Multiple Importance Sampling	21
	4.7 Bi-directional Path Tracing	23
	4.8 Metropolis Light Transport	25
5	Rendering Effects	30
	5.1 Surface-to-surface Reflection/Refraction	30
	5.2 Volume Rendering	31
	5.3 Subsurface Scattering	34
	5.4 Environment Map and Material Texture	36
6	SIMD Optimization	39
	6.1 Data Structure Rearrangement	39
	6.2 Thread Divergence Reduction	41
	6.3 Thread Compaction	42
	6.4 A More Efficient Ray-Triangle Intersection Algorithm	44
	6.5 GPU SAH Kd-Tree Construction	45
7	Benchmarking	50
8	Conclusion	54
	8.1 Summary	54
	8.2 Limitations & Recommendations for Further Work	55
Bibliography		iv
Appendix		vi

Chapter 1 Introduction

1.1 Background Information

Polygon rasterization method has been the de-facto standard of real-time graphic generation technique of video gaming in past few decades, where ray tracing related methods are still mainly used in off-line rendering of animation films and industrial designs. However, recent years have witnessed a rapid growth in the capability of real-time ray tracing with the advent of General Purpose GPU (GPGPU) and associated programming interfaces like CUDA and OpenCL. Nowadays, it is possible to ray-trace complex scene without global illumination in real-time on high-end GPUs. Because of its theoretical straightforwardness of dealing with complex optical effects and the huge potential of performance growth thanks to the performance scalability of GPU which responds directly to Moore's law without the power wall faced by CPU, ray tracing related methods have been considered as the standard graphic rendering technique of the future.

However, to achieve photorealistic effects which, in rasterization based graphics, are implemented by heavy use of textures, a ray tracing model must include global illumination beyond the simplified light paths defined in Whitted ray tracing. Some ancillary methods like radiosity and distribution ray tracing have been added into the original Whitted framework to enhance the lighting effect, yet each of them targets at a specific subset of the general global illumination and the combination of them is not exhaustive of all possible light paths. Path tracing, which samples all possible light paths using Monte Carlo methods and the rendering equation has come to the rescue. Fundamentally expanding the scope of traditional ray tracing, path tracing can naturally generate authentic global illumination effect within its theoretical simplicity. Yet, it requires thousands of samples per pixel to reduce the noise on picture below human perception and is therefore used mostly in offline rendering tasks like film production. Interactive refinement method which averages the result among consecutive frames (each frame takes an extra sample for every pixel and is accumulated to the result of last frame if the camera is still) has been adopted in some real-time path tracing demos. However, most of these demo programs are hardly

optimized and only contains a very limited subset of rendering effects, which gives reason to the implementation of a versatile, robust, and highly efficient real-time path tracer.

1.2 Objectives of Project

The main objective of this project is to explore the capability and performance of combination of the power of current GPGPU with existing and new path tracing algorithms in the task of real-time path tracing. To achieve this, a variety of different factors that determines the efficiency of path tracing are studied, amongst which spatial acceleration structures, sampling algorithms, and single-instruction-multiple-data (SIMD) optimization are most important. Since the variable (lighting configuration, scene geometry and material) and measure (frame rate, convergence rate) of path tracing performance are both multi-dimensional, optimization concepts are provided in a case-by-case analysis. A standalone program is written to demonstrate these concepts. To guarantee that the concepts are applicable to general and complex cases, a considerably large subset of all functionalities found in state-of-art path tracers is integrated into the program which includes PBR (physically-based rendering) material and participating media. Besides, bi-directional path tracing and Metropolis light transport are also studied to deal with cases containing difficult lighting configuration and to improve the rendering quality under same time constraints.

1.3 Layout of Thesis

The main content of the thesis will be divided into 6 parts as Chapter 2 – Chapter 7. In Chapter 2, we will have an overview of the workflow of our path tracer, followed by the spatial acceleration structures including SAH-based Kd-Tree and BVH as the first studied factor of optimization in Chapter 3. In Chapter 4, the rendering equation will be reviewed with normal Monte Carlo sampling algorithm, after which advanced sampling techniques - multiple importance sampling, bi-directional path tracing and Metropolis light transport based on Markov Chain Monte Carlo method will be studied in response to difficult rendering cases and noise reduction. Before the discussion of SIMD optimization in Chapter 6, several important shading models including surface-to-surface BSDF, ray-marching volume rendering and a simplification of subsurface scattering will be introduced due to their close relationship with sampling methods. In particular, I will propose a parallel

SAH-based Kd-Tree construction algorithm that is suitable for current GPGPU in Chapter 6. In Chapter 7, benchmark methods will be introduced which carry out the comparison between my path tracer, NVIDIA's Optix path tracing demo, and Cycles Renderer – a free mainstream path tracer.

Chapter 2 Overview of Software Workflow

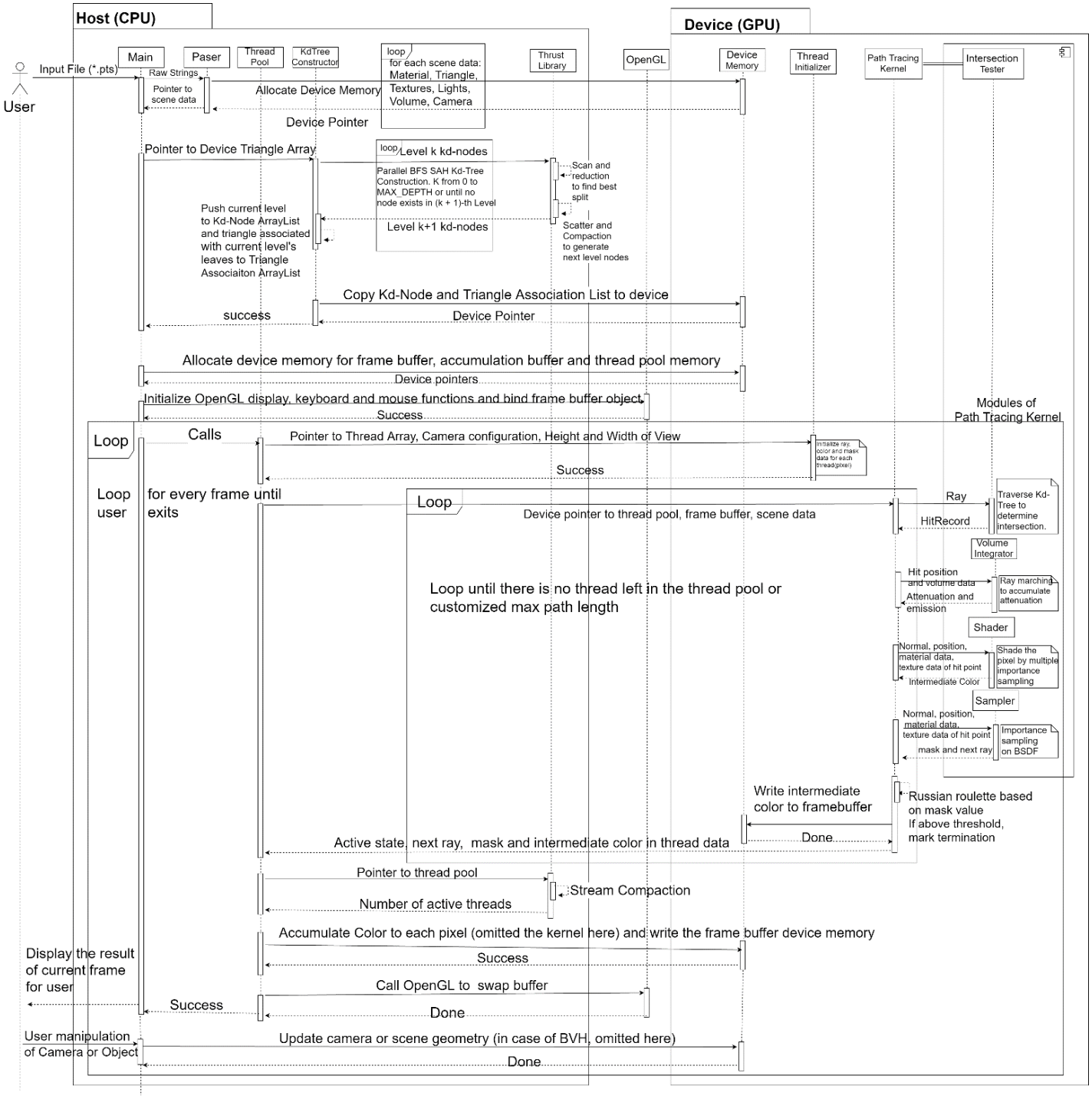


Figure 1 UML flowchart of our path tracer

The diagram above shows the simplified workflow of the proposed path tracer. Note that modules for metropolis light transport and bi-directional path tracing are not included in this diagram, which only describes the normal unidirectional path tracing. However, one can easily modify the diagram to get the versions for metropolis light transport and bi-directional path tracing, which shares most of the processes. Also, it only shows the case when kd-tree is used as the spatial acceleration structure. In fact, BVH is also implemented especially for user manipulation of scene geometry.

An obvious characteristic of the displayed architecture is that modules seem to be evenly distributed on CPU and GPU. In fact, GPU consumes most of the living time of the program, as CPU is only responsible for some preprocessing work like handling I/O, parsing, invoking memory allocation, calling CUDA and OpenGL API and the coordination between thread pool and GPU kernel. Despite its frequent involvement in the path tracing task (it appears between every two recursion levels in a frame), CPU occupies only a fractional of the time. Notwithstanding its conciseness, the diagram shows all 3 optimization factors in path tracing. Kd-tree, which is constructed on GPU in this project, is highly optimized by the "short-stack" traversal method which will be introduced later. Multiple importance sampling, as a generalization of single importance sampling, can be used for rendering glossy surface under strong highlights to reduce the variance. Thread compaction, a crucial method for increasing proportion of effective work and memory bandwidth, representative of the SIMT optimization, is shown at the bottom of the diagram. A thread pool is maintained to coordinate the work of thread compaction.

After initialization, the program spends all of the rest of the time in two loops, the outer of which uses successive refinement algorithm to display image in real-time, and the inner of which executes a path tracing recursion level for all threads in every iteration. Between two inner iterations, thread compaction is used to maintain occupancy as mentioned before. Between two outer iterations, any user interaction is processed by CPU. After updating corresponding values in device memory, OpenGL API is called to swap frame buffers.

It is noticeable that the Thrust library is also an important component of our workflow. Developed by NVIDIA, Thrust is a C++ library for CUDA providing all parallel computing primitives like map, scatter, reduce, scan and thread compaction. As a high-level interface,

Thrust enables high performance parallel computing capability while dramatically reduces the programming effort (NVIDIA, 2017). Rather than “reinventing the wheel”, we use thrust for all parallel computing primitives required for GPU kd-tree construction and thread compaction in our program due to the proven efficiency it provides and the flexibility of its API.

Overall, the diagram shows a very macroscopic outline of the software structure, whose detail will be introduced in the following chapters. In addition, some limitations and recommended improvements will be addressed in the final chapter.

Chapter 3 Spatial Acceleration Structure

3.1 Choice of SAS

The naïve implementation of ray tracing related algorithms iterates through the set of all primitives in the scene and checks ray-primitive intersection for each, which is very time-consuming (linear to the number of primitives) and is a severe bottleneck in performance when the number of primitives gets high. In reality, different spatial acceleration structures (SAS) are applied to solve the problem. They generally improve the speed to logarithmic time and therefore can make interactive ray tracing for complex or even dynamic scene. Octree, BSP (binary space partitioning), BVH (bounding volume hierarchy) and kd-tree are some representatives of the SAS. The SAS generally divide the scene or mesh into recursive sub-spaces which often has a tree-like structure. Among them, octree and BSP are the type of solution which chooses split position in a fixed routine. For example, a typical octree always chooses the center of the space to divide it into 8 sub-spaces. Since they are indiscriminate to the specific geometry that the scene has, they generally exhibit lower efficiency than BVH or kd-tree. BVH or kd-tree, on the other hand, uses some heuristics to determine the partition position based on the specific scene geometry. In terms of the efficiency of BVH and kd-tree, Vinkler et al. (2014) has shown that kd-tree has higher performance for complex scenes than BVH while BVH defeats kd-tree for simple to moderately complex scenes. Considering this, both structures are implemented for the freedom of choice with respect to different kinds of scenes. In addition, BVH is necessary for real-time ray tracing against dynamic scene geometry with complex moving meshes as it maintains the interior hierarchy of the mesh and only updates exterior hierarchy which is usually much simpler to do.

3.2 Surface Area Heuristics

The construction of kd-tree and BVH depends on choosing one dimension and the split position in that dimension in every iteration. A naïve solution is to cycle through the 3 axes and choose the space median every time, giving no better performance than octree. A popular mechanism is Surface Area Heuristics (SAH) (Wald & Havran, 2006), which is based on the greedy algorithm to find a local optimum based on the surface areas of the

two child nodes in every step. Instead of finding the global minimum cost which is practically infeasible as number of possible trees grows exponentially with scene complexity, SAH assumes all the primitives in child nodes of a particular step are in leaves, giving the formula of the expected cost of a particular split:

$$C_v(p) = K_T + K_I\left(\frac{SA(V_L)}{SA(V)}|T_L| + \frac{SA(V_R)}{SA(V)}|T_R|\right),$$

where $SA(V_x)$ is the surface area of volume x, K_T, K_I correspond to the cost of a traversal step and an intersection step, and T_L, T_R are number of primitives in left and right child node. According to probability theory, $\frac{SA(V_L)}{SA(V)}$ gives the chance of uniformly distributed rays hitting the left node based on the reasonable assumption that the distribution of rays tends to not follow any certain pattern with the number of ray bounces increasing and geometry of the scene varies. On the other hand, although treating both nodes as leaves overestimates the real cost, the strategy works well in practice (Wald & Havran, 2006). Another advantage of SAH is determination of when to stop splitting is easy, as one can compare the cost of splitting and not splitting directly from the above formula.

Despite its high performance in traversal, the construction of the SAH-based tree is not a trivial task and can cause severe initialization overhead for complex scene. We will use the construction of SAH-based kd-tree as an example in the following sections of the chapter. Wald & Havran introduced an O(N log N) SAH-based kd-tree construction algorithm which beats the $O(N^2)$ or $O(N \log^2 N)$ time of tradition algorithms. However, the construction time can still be largely compressed with the computing power of today's GPGPU. Since the GPU construction only parallelizes the algorithm without changing its key ideas, this topic will be discussed in Chapter 6 – SIMT optimization. In this chapter, we will focus on issues of the performance of SAS in path tracing.

3.3 Triangle Clipping in Kd-Tree Construction

An important issue in kd-tree construction is the necessity to clip triangles which span across the splitting plane in each level for accelerating intersection. In their 2006's paper, Wald & Havran suggested that on average, for a kd-node with N triangles, there are $O(\sqrt{N})$ triangles spanning across a splitting plane. Normally, we check whether the ends on current

chosen splitting axis of the triangle's bounding box are in distinct sides of the splitting plane to determine whether to add it in both of the child nodes. However, it may be the case that the triangle does not overlap one of the child node, even though in one dimension it does. Such error will be accumulated to a situation that the whole triangle lies outside of its node's bounding box. With the increase of kd-tree level, $\frac{\sqrt{N}}{N}$ will be increasingly close to 1, which means at leaf level, one would expect many false positives to occur in intersection test, unnecessarily increasing the time for intersection. In addition, adding spanning triangles to both sides unnecessarily increases the workload of construction as one has to test the vertices against the boundary of the bounding box in current kd-tree level to avoid choosing a splitting plane outside the bounding box.

Since it is convincing from the analysis above that clipping triangle does has an obvious boost on intersection performance, which is often a bottleneck for path tracing, we offer some test cases to quantize the rate of performance improvement. After that, we will explain how to clip the triangles, which is a relatively simple task without the need of importing third-party libraries.

Model Name	Suzzane	Stanford Bunny	Armadillo	Happy Buddha
Sample Image				
Face Count	500	69,451	345,944	1,087,716
FPS w/o clipping	39.73	33.11	27.30	25.33
FPS w clipping	42.63	36.12	28.25	26.56
Speedup	7.3%	9.1%	3.5%	4.9%

Table 1 Speedup of triangle clipping for different models

From the table, we can observe that clipping triangles results in a speedup from 3.5% to 9.1% for different mesh complexity, which is not very drastic but obvious enough to confirm the effect of triangle clipping.

We will now illustrate how to clip arbitrary triangle against box with an example where a triangle is clipped to a pentagon. In Figure 2 below, we determine the intersection between any pair of vertices that lies in different sides of each of the 6 faces of the cuboid. Then, in the plane spanned by each of the 6 faces, we calculate the intersections the line extended by the two intersection points (if there is two), which may be zero, one or two in number. Any intersection will become a vertex of the clipped polygon. If the end vertex itself lies inside or on the border of the box's face, it also become a vertex of the clipped polygon. Notice that the process is trivially parallelizable for the 6 faces, except for the memory write, i.e. expanding current bounding box to contain the new vertex. If the construction is running on GPU, parallelizing on 6 faces can decrease considerable amount of time for the clipping stage in each level, which does not process too many triangles and is solely occupying the GPU.

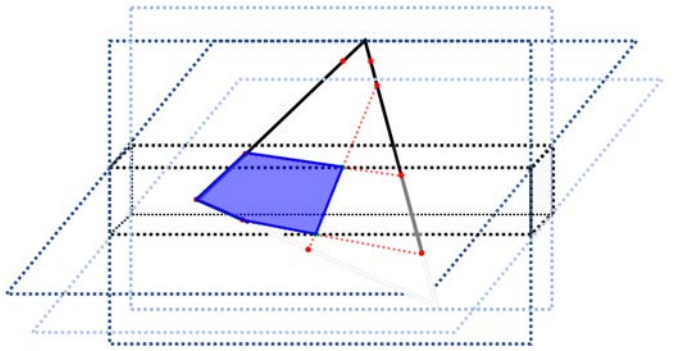


Figure 2 A triangle clipping example

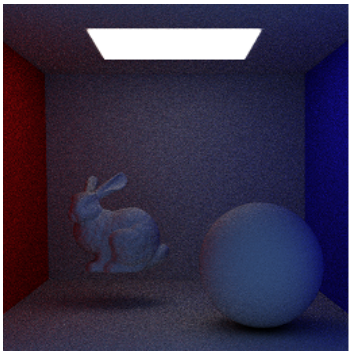
3.4 Kd-tree Traversal

For the traversal of kd-tree, the standard CPU algorithm with stack which stores backside nodes cannot be directly applied to GPU. First, the stack needs to be implemented in fixed length array which guarantees coalesced memory access and reduces memory read and write instead of using linked list or dynamic array implementation of stack. Second, size of the stack item should be compressed as small as possible as complex scene with dozens of kd-tree levels will require the stack to be allocated in local memory instead of the thread

registers with very limited capacity. Foley & Sugerman (2005) introduced two stackless traversal algorithms called kd-restart and kd-backtrack. However, without the proper priority information stored for traversal, these algorithms require modification of the traversal path which brings extra time and space complexity: kd-restart directly goes into the nearest leaf and restarts from the root with ray range propelled forward and kd-backtrack stores extra node data to improve traversal restart efficiency as it can restart from a node's parent. Meanwhile, the worst case of kd-restart degenerates to linear.

A neat solution proposed by Santos et al. (2012) adopted a "short stack" method. Instead of storing 12 bytes as in standard algorithms (4 bytes for node address, 4 bytes for near ray distance (tnear), 4 bytes for far ray distance (tfar)), they discovered that tnear can be derived in the traversal process and it only updates when the traversal finishes checking a leaf, giving an "8-bytes" stack algorithm. The advantage of "8-bytes" stack is not only fewer local memory required, but faster memory access thanks to the fact that an 8-byte load is faster than a 12-byte loads in local memory in CUDA architecture.

By combing a SAH construction of kd-tree and a "short stack" traversal, my SAS has the optimal performance comparing with other combinations. Below is the experiment data of different combinations of construction and traversal methods on different scene (Figure 3). Notice that these data are the result of some SIMD optimization applied to the data structure, which I will discuss and compare the performance with non-optimized ones in Chapter 6.



	SAH	Space Median
8-byte stack	6.235	8.056
Kd-restart	10.139	15.804
Kd-backtrack	30.516	45.240

Time (seconds) to render 64 samples/pixel
"bunny&ball" scene in 512x512

Figure 3

3.5 SAH-based BVH

As mentioned before, BVH is a crucial component for real-time ray tracing against dynamic scene geometry. Similar to kd-tree, the probabilistic analysis applies to the decision of splitting plane in each tree level, which naturally leads to the fact that greedy choice of local optimum gives the best available algorithm for optimizing traversal cost. The only difference between kd-tree and BVH in terms of surface area heuristic is – BVH is a hierarchy of objects and kd-tree is a hierarchy of subspaces. Therefore, refitting bounding box is necessary when dividing the node into child nodes, which turns out to be a time-consuming bottleneck in construction.

Unlike kd-tree construction which uses a dynamic array or vector to store all triangle events, BVH construction needs to maintain a binary search tree for each dimension. The shrinking side of the refitting process requires us to search and delete the events that are recently switched to the expanding side and add them to the BST of that side. Initialization of the BSTs costs O (N log N). Each check of splitting position costs O(log N) and the whole process of best plane determination costs O (N log N), leading to a O (N $\log^2$ N) total time complexity. Even worse, maintaining binary search tree for three dimensions implies a big constant of 3. For this reason, the construction of SAH-based BVH often causes serious overhead latency in complex cases. Fortunately, a simplification method which equally divides the space into a small number of buckets was proposed by Pharr & Humphreys (2011) in their famous *Physically Based Rendering* book. Partitions are then only considered at bucket boundaries, which gives much more efficient construction while remaining as effective in traversal as the original method. It is easy to figure out that the whole construction only requires O(N log N) time since bucket number is a constant. Meanwhile, a binned partition implies easier and more efficient parallelization on GPU. Due to time limit, a GPU construction for BVH is not implemented, as there exist well known parallel binned BVH construction methods.

The traversal of BVH is easier to implement than kd-tree but less efficient in performance. Since we cannot guarantee any deterministic spatial order of BVH nodes as they may overlap each other in any form, we cannot terminate traversal after we find a hit in leaf. Although the child nodes of a BVH node are also stored in a “front-back” order. It only

indicates the spatial order of the centroid of two bounding boxes as in construction the decision of affiliation of triangle is based on the side of its centroid. It is entirely possible that the “back” BVH node contains a nearer hit than the “front” node. Nevertheless, such probability is not high. In most cases, the “back” BVH node will not have any intersection in the trimmed range after intersection is done for the “front” BVH node, after which it will be popped from stack. If then the stack is checked to be empty, the function returns the nearest intersection if there is any.

3.6 Automatic BVH Refitting

A simple solution of tracing dynamic scene geometry is to recursively refit the local BVH whenever intended object moves beyond its parent’s bounding box. If tree levels outside the intended object is much less than tree levels inside or movements of the object is spatially limited, such method has a very low time cost. Attention must be paid to the fact that shrinking refitting is also necessary when object moves towards the original position, for which we can store a record of the moving direction of current frame as bitmask of 3 axes. If the bounding box of moving object in last frame borders its parent or ancestor with respect to any of the current dimension of moving read from the bitmask, we perform a shrinking refit. Also, for every movable object, a translation vector is stored to be used as an offset in triangle intersection. However, when assumption of less exterior tree level does not hold or there is violent movement of objects, we need to consider alternative methods other than the purely refitting. A combination of splitting, merging and rotation operations can be performed on tree structure (Kopta et al., 2012), which massively increases the rendering speed for complex animated scenes as it avoids structural degeneration in naïve refitting.

However, such method also has its limitation. When most of the objects in the scene are animated (e.g. particle system), update of BVH is serialized due to necessary atomic operations for many threads changing the boundaries of the same bounding box. In this situation, it is better to rebuild the BVH rather than modify it.

Chapter 4 Sampling Algorithm

A key feature of path tracing that differentiates it with normal ray tracing is that it is a stochastic process (provided that the random numbers are real) instead of a deterministic process. Normal path tracing depends on Monte Carlo algorithm which gradually converges the result to the ground truth as the number of samples increases. Theoretically, one can uniformly sample all paths to converge to the correct result. However, given limited hardware resource and time requirement, we need to adapt the brute Monte Carlo algorithm by various strategies for different cases. The following sections will introduce the rendering equation we need to solve in path tracing and some most popular sampling methods.

4.1 The Rendering Equation

The rendering equation introduced by Kajiya (1986) defines the radiance seen from a point $\boldsymbol{x}$ in the reflection direction ω_{ref}, i.e. view vector in ray tracing's grammar as a result of reversibility of light path:

$$L_{ref}(\boldsymbol{x}, \omega_{ref}) = L_e(\boldsymbol{x}, \omega_{ref}) + \int_{\Omega} f(\boldsymbol{x}, \omega_{in} \rightarrow \omega_{out}) L_{in}(\boldsymbol{x}, \omega_{in}) cos\theta_{in} d\omega_{in} ,$$

which is the emittance of the point itself plus the reflectance caused by all incident radiance summed from the surrounding hemisphere. This is a physically correct model of global illumination considering only surface to surface reflection. In computer graphics, this is usually mapped into a recursion where the integral is decomposed into randomly picked samples. For path tracing, all possible light paths from the set of light rays within a pixel are sampled individually from random positions within the pixel whose results are then averaged. Upon hitting a surface, only one secondary ray is shot for each sample. It intuitionally follows that the secondary ray must be generated with some probability mechanisms, which is defined by BSDF (bi-directional scattering distribution function, union of BRDF (reflectance) and BTDF (transmission)) $\rho(\boldsymbol{x}, \omega_{in} \rightarrow \omega_{out})$ with respect to the surface property. However, there are some problems. Given limited number of samples, how do we choose a proper sampling strategy to maximize the image quality? Given a

specific BSDF, how to translate it into an algorithm that fits into the sampling strategy we use?

4.2 Stratified Sampling vs. Anti-aliasing Filters

To generate samples within pixels, a naïve solution is uniform sampling. In CUDA device code, the function curand_uniform(seed) can generate 1D uniform pseudorandom samples from 0 to 1. However, the uniformly distributed samples tend to form clusters, producing high noise level. A common way to overcome this is stratified sampling, which divides a pixel into $D \times D$ grids and takes uniform samples of same number within each grid. Theoretically, it reduces the error of estimation from $O\left(\frac{1}{\sqrt{n}}\right)$ to $O\left(\frac{1}{n}\right)$, where n is the number of samples. A problem of stratified sampling is that it is not suitable for successive refinement required in real-time rendering. When using successive refinement, usually one sample is taken for every pixel in each frame. If we want to have minimum aliasing effect in any given frame, the former samples must at least not follow any certain fixed pattern, which is not possible for stratified sampling which must use at least $D \times D$ samples as a unit. Therefore, we want to find a solution having both low noise level and successive refining capability.

In the famous 99 lines of C++ implementation of path tracing SmallPT, Beason (2007) applies a tent filter to the uniform random samples which shift more samples towards the center of the pixel. In my test, this method produces same image quality given same sampling number as stratified sampling does with the ability of successive refinement. In fact, it approximates the sinc function, the ideal anti-aliasing filter (Figure 4). There are actually other choices of approximation with higher quality such as bicubic filter and Gaussian filter. However, these filters have much higher computation overheads while the tent filter is a more practical choice in real-time rendering.

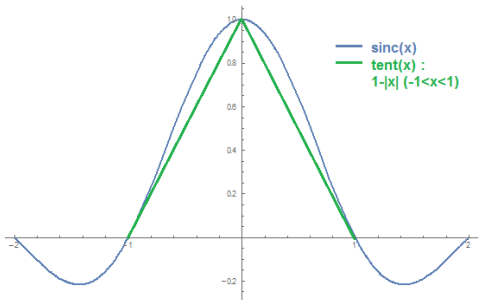


Figure 4. Comparison between sinc function and tent function

In practice, a combination of anti-aliasing filtering and stratified sampling is more desirable in order to further decrease the noise level. In our implementation, every pixel is divided into 4 subpixels and we generate rays by anti-aliasing filtered uniformed sampling and average the returned color for all 4 subpixels. Counterintuitively, a frame executing such process actually costs considerably less time than 4 frames without subpixel division in our implementation. In Chapter 6, we will explain thread compaction, which is responsible for this effect.

4.3 BSDF and Importance Sampling

Importance sampling is an effective method for solving rendering equation in high convergence rate. Basically, importance sampling chooses samples by a designed probability density function and divides the sample value by p.d.f. to return the result. If the designed p.d.f. turns out to be proportional with the values, variance will be very low. Since the sample value is determined by $f(\boldsymbol{x}, \omega_{in} \rightarrow \omega_{out}) L_{in}(\boldsymbol{x}, \omega_{in}) cos\theta_{in}$, both irradiance and BSDF decides the p.d.f. $p(\omega_{out})$ of the surface sample. Importance sampling BSDF is a more trivial task than importance sampling the spatial distribution of incoming radiance. As long as a BSDF formula (Lambertian, Cook-Torrance, Oren-Nayar, etc.) and corresponding surface characteristics are provided, one can calculate $\frac{f(\boldsymbol{x}, \omega_{in} \rightarrow \omega_{out})}{p_{bsdf}(\boldsymbol{x}, \omega_{in} \rightarrow \omega_{out})} L_{in}(\boldsymbol{x}, \omega_{in}) cos\theta_{in}$ as the result. However, distribution of $L_{in}(\boldsymbol{x}, \omega_{in})$ is harder to calculate in general case. For indirect lighting on the surface point, it is impossible to know the distribution of the incoming radiance, which is a chicken and egg paradox. For direct lighting where $L_{in}(\boldsymbol{x}, \omega_{in}) = L_e$, different methods can be applied to find the p.d.f. With a simplified lighting model like point light or area light, the distribution of incoming radiance is rather explicit. However, when image-based lighting is used (e.g. environmental

map or sky box), advanced techniques are required to perform importance sampling efficiently. The following sections will all focus in the case of having explicit lighting models since it is easier to exemplify how to utilize the p.d.f. of incoming radiance. Discussion for image-based lighting will be continued in the next chapter. Since combing the effect of two p.d.f. requires not only sampling on the surface point, but the lighting model or lighting image as well. A technique called multiple importance sampling will be introduced in Section 4.5.

4.4 Next Event Estimation

For direct lighting with explicit model, lighting computation can be directly done without shooting ray to the lights, which is called next event estimation. Literally, it accounts for the contribution of what may happen in the next iteration. For ideal diffuse surface, whose BSDF is spatial uniform (usually expressed by Lambertian model), this task is very simple. One only needs to sample a point in one of the lights. Take diffuse area light (emission of a point is uniform in all directions) as an example, if the emission across the light emission surface is the same, one only needs to uniformly sample the shape of the light. Otherwise, usually there is an existing intensity distribution to use. After that, to convert the area measure of p.d.f to solid angle measure, from formula $p(\omega)d\omega = p(x)dA(x)$ (Veach, 1997) we can derive $p(\omega) = p(x)\frac{dA}{d\omega} = \frac{p_e(x)}{Area}\frac{||x_{light}-x_{surface}||^2}{|N_{light}\cdot L|}$, which can be used as the p.d.f for incoming radiance. Given that Lambertian surface has a BSDF of $\frac{C_{diffuse}}{\pi}$, the final color can be expressed in the simple formula $\frac{C_{diffuse}L_e(x)|N_{surface}\cdot L||N_{light}\cdot L|Area}{\pi||x_{light}-x_{surface}||^2 p_e(x)}$.

For diffuse area lights of simple geometric shape like triangle, uniform sampling can be trivially done by using barycentric coordinates. Apart from light sampling, we also need to shoot a shadow ray from the surface point being shaded to the sampled light point. Since ray-triangle intersection test is a bottleneck in ray tracing for reasonably complex scenes, this means we will almost halve the frame rate. However, the benefit of next event estimation totally worth the costs it takes. Figure 5 is an example picture comparing the convergence rate of diffuse reflection of two balls under highlight with and without next event estimation, where 16 samples are taken for each pixel for both sampling methods. Although the frame rate for rendering with next event estimation is only 60% of that

without next event estimation, the noise level of the former is dramatically lower than the latter.

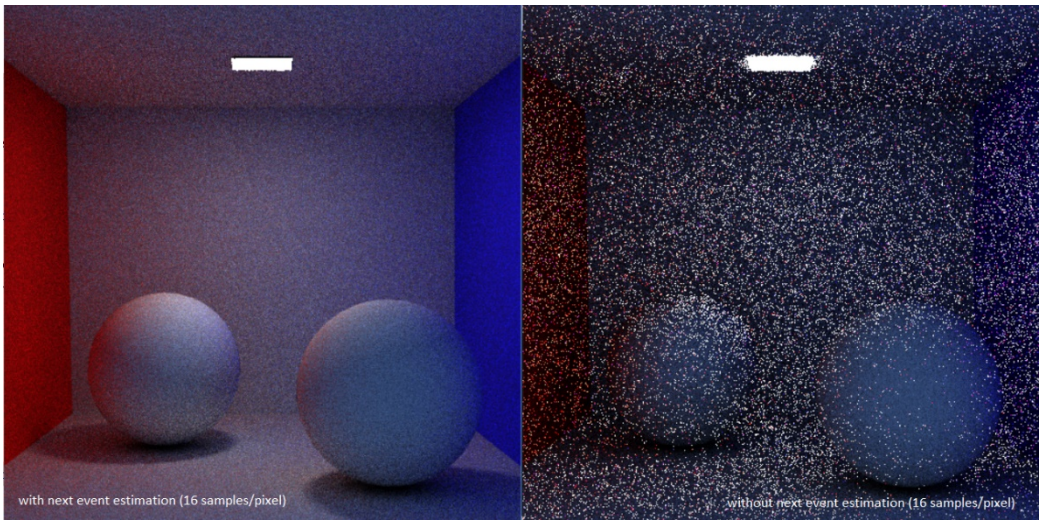


Figure 5. Comparison between same scene of high dynamic range with and without next event estimation

However, this example only shows the case where the shaded surface has a uniform BSDF. For surfaces with general non-uniform BSDF like glossy surface in the Cook-Torrance microfacet model, sampling the light is inefficient for variance reduction. An extreme case is the perfect mirror reflection, whose BSDF is a delta function. Since only the mirrored direction of the view vector w.r.t the surface normal contributes to the result, sampling from a random point in the light model has zero probability to contribute. We would naturally want to sample according to the BSDF. A general case is, a glossy surface whose BSDF values concentrate in a moderately small range and the lighting model occupies moderately large portion of the hemisphere around the surface being shaded. As a rescue for general cases, multiple importance sampling will be introduced later. However, if we want to have a balance between quality and speed, next event estimation can be mixed with direct path tracing for different BSDF components. Especially for the case of surface material with only diffuse and perfectly specular reflection, doing a next event estimation by multiple importance sampling would be redundant. Instead, if the BSDF component in current iteration is diffuse, we will set a flag to avoid counting in the contribution if we hit a light in next iteration. Otherwise, such flag will have a false value, allowing the radiance of the light hit by the main path to be accounted into. For determining which BSDF

component to sample, we will use a “Fresnel switch”, which will be introduced in the next section.

4.5 Fresnel Switch and The Russian Roulette

Physically, there are only reflection and refraction when light as an electromagnetic wave interacts with a surface, the ratio of which is determined by the media’s refractive indices in two sides of the surface and the incidence angle of the light. The original formula is actually different for s and p polarization component in the light ray. However, in computer graphics, we normally treat the light as non-polarized. Under this assumption, Schlick’s approximation (Schlick, 1994) can be used to calculate the Fresnel factor: $R(\theta) = R_0 + (1 - R_0)(1 - cos\theta)^5, where\ R_0 = (\frac{n_1 - n_2}{n_1 + n_2})^2$. However, in the standard PBR workflow, the refractive index is usually only provided for translucent objects. Although we can look up for the refractive index of many kinds of material, for metals and semiconductors, the refractive index n is a complex number, which complicates the calculation. Since the refractive index of the metal indicates an absorption of the color without transmission, specular color is used to approximate the reflection intensity in normal direction. In practice, there is usually an albedo map and a metalness map for metallic objects. The metalness of a point determines the ratio in interpolation between white color and albedo color. In the case of metalness equal to 1, the albedo color will become the specular color in reflection, as what we refer to as the color of metal actually suggests how different wavelengths of light is absorbed rather than transmitted or scattered in the case for dielectric material.

Briefly, if there is an explicit definition of index of refraction, we will use that for R_0 calculation of the material. Otherwise, we interpolate the albedo color provided in material map or scene description file with the white specular color meaning complete reflection of light according to the metalness of the material, as defined in normal PBR workflow. However, considering the fact that dielectric material also absorbs light to some degree, the specular color can be attenuated by some factor less than 1 to provide a more realistic appearance.

After R_0 is calculated, we can calculate the Fresnel factor by substituting the dot product of view vector and the surface normal into the Schlick's formula. Notice that there is a power of 5, which is better calculated by brute multiplication of 5 times than using the pow function in the C++ or CUDA library for better performance. The Fresnel factor R indicates the ratio of reflection. In path tracing, this indicates the probability of choosing the specular BRDF to sample for the next direction. Complementarily, T = 1 – R expresses the probability of transmission.

In our workflow, diffuse reflection is also modeled as a transmission, which will immediately be scattered back by particles beneath the surface uniformly in all directions, which is usually modeled by Lambertian BSDF. However, to be more physically realistic, we can refer to Ashikhmin and Shirley's model (2000) which models the surface as one glossy layer above and one diffuse layer beneath with infinitesimal distance between. The back scattering in realistic diffuse reflection happens at the diffuse layer as a Lambertian process, after which the reflected ray is attenuated again by transmitted across the glossy surface, implying less contribution of next ray in near tangent directions. For energy conservation, Ashikhmin and Shirley also include a scaling constant $\frac{28}{23}$ in the formula. Therefore, the complete BSDF is :

$$f(\boldsymbol{x}, \omega_{in} \rightarrow \omega_{out}) = \frac{28}{23}\frac{C_{diffuse}}{\pi}(1 - R_0)(1 - (1 - \frac{n \cdot \omega_{in}}{2})^5)(1 - (1 - \frac{n \cdot \omega_{out}}{2})^5).$$

Russian roulette is used to choose the BSDF component given the Fresnel factor and other related material attributes. If the generated normalized uniform random number is above the Fresnel threshold R, next ray will be transmission or diffuse reflection. Again, the "translucency" attribute of the material will be used as the threshold for determining transmission or diffuse reflection, which is actually an approximation of general subsurface scattering, which will be discussed in next chapter. The Fresnel switch guarantees we can preview the statistically authentic result in real-time. However, the thread divergence implies a severe time penalty in GPU. Suggestion will be given in the SIMT optimization analysis in Chapter 6.

The Russian Roulette is also responsible for thread termination. Since any surface cannot amplify the intensity of incoming light, the RGB value of the mask

($\frac{f(\boldsymbol{x},\omega_{in}\rightarrow\omega_{out})}{p_{bsdf}(\boldsymbol{x},\omega_{in}\rightarrow\omega_{out})}cos\theta_{in}$) is always less than or equal to 1. The intensity of the mask (which is the value of the largest component of RGB, or the value in HSV decomposition of color) is then used as the threshold to terminate paths. To be statistically correct, the mask is always divided by the threshold value after the Russian Roulette test, which is a very intuitive process – if the reflectance of the surface is weak, early termination with value-compensated masks is equivalent to multiple iterations of normal masks. Such termination decision greatly speeds up rendering without increasing the variance.

For generating photorealistic result, it is possible to use the Russian Roulette solely to determinate termination without setting a maximum depth. However, considering the extreme case where the camera is inside an enclosed room and all surfaces are perfectly specular and reflect all lights, there is still a need to set a maximum depth.

4.6 Multiple Importance Sampling

Returning to our question of next event estimation or direct lighting computation for general surface BSDF, the technique called multiple importance sampling was introduced by Eric Veach in his 1997's PhD dissertation. Basically, it uses a simple strategy to provide a highly efficient and low-variance estimate of a multi-factor integral mapped to a Monte Carlo process where two or more sampling distributions are used, usually in different sampling domains. Given an integral $\int f(x)g(x)dx$ with two available sampling distributions p_f and p_e, a simple formula given to as the Monte Carlo estimator $\frac{1}{n_f}\sum_{i=1}^{n_f}\frac{f(X_i)g(X_i)w_f(X_i)}{p_f(X_i)}+\frac{1}{n_g}\sum_{j=1}^{n_g}\frac{f(Y_j)g(Y_j)w_g(Y_j)}{p_g(Y_j)}$, where n_f and n_g are number of samples taken for each distribution and $w_f(X_i)$ and $w_g(X_i)$ are specially chosen weighting functions which must guarantee that the expected value of the estimation equals the integral $\int f(x)g(x)dx$. As expected, the weighting functions can be chosen from some heuristics to minimize the variance.

Veach also offers two heuristic weighting functions: balance heuristic and power heuristic. In a general form $w_k(x)=\frac{(n_kp_k(x))^\beta}{\sum_i(n_ip_i(x))^\beta}$ where k is any particular sampling method, balance heuristic always takes β = 1 while power heuristic gives the freedom to choose any

exponent β. Veach then uses numerical tests to determine that $\beta = 2$ is a good value for most cases.

In order to verify the practical result of multiple importance sampling and the effect of the choice of weighting function, some test cases were performed and sample images of the results are listed below.

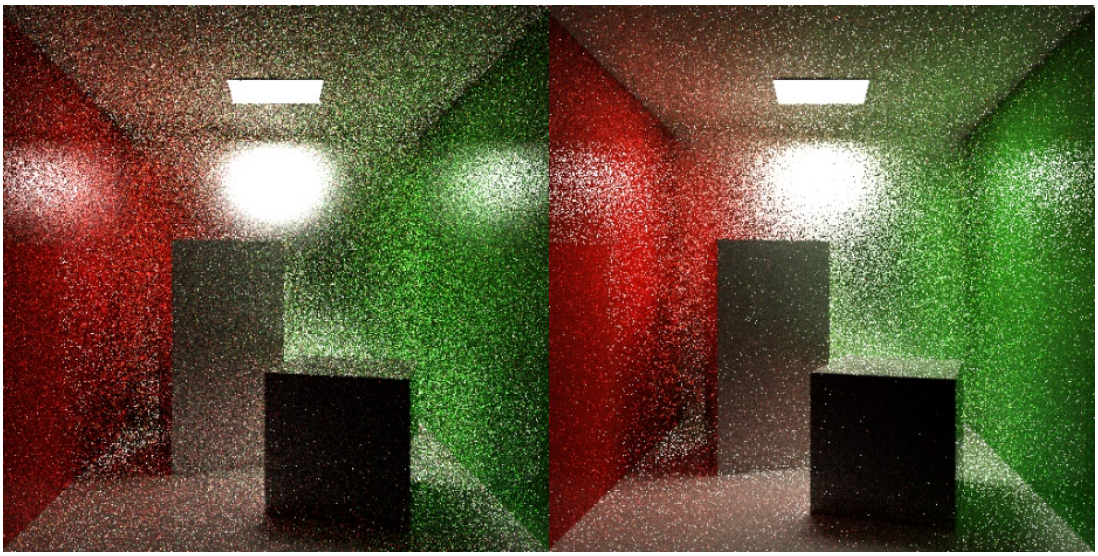


Figure 6 Left: multiple important sampling Right: single importance (light) sampling

The first test case compares the result of multiple importance sampling (both from BSDF and light) and single importance sampling (only from BSDF). In each frame, a path is traced for each pixel. Although rendered only 10 frames, the MIS result clearly displays the shape of the reflection of the strong area light on the rough mirror behind the boxes and its further reflection on the alongside walls. In contrast, the non-MIS result generated by 100 frames still has a very noisy presentation of the reflected shape of the highlight. Note that for reflection on the floor and ceiling, which has a low roughness, non-MIS in frame 100 still has a lower noise level than that of MIS in frame 10. Since most contribution to the color comes from samples generated from BSDF, the strength of multiple importance sampling diminishes, which is also the case when the BSDF is near uniform.

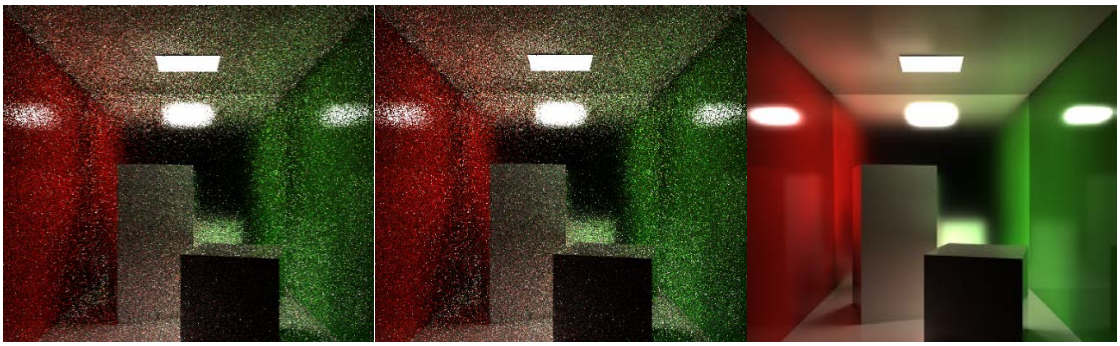


Figure 7 Left: balance heuristic Middle: power heuristic Right: ground truth

Another test case aims at comparing the effect of balance heuristic and power heuristic. The images in Figure 7 above show the result of rendering the same Cornell box scene with moderately rough back mirror in 10 frames for both methods. Without very carefully inspecting the images, it is nearly impossible to observe any difference between two images. However, noise level is indeed lower when using power heuristic. Intuitively, if one carefully looks at some dark regions in the picture like the front face of the shorter box, an observation that many noise points are brighter in the image generated by balance heuristic (Since both cases use the same seed for random number generation, it is possible to compare noise point at same pixel position). However, we also offer numerical analysis to compare the percentage of difference between the two images and the ground truth (rendered by metropolis light transport in 80000 frames). The result of calculating the histogram correlation coefficient with the ground truth for each image shows that image generated by power heuristic has a value of 0.7517, larger than the image generated by balance heuristic which has a value of 0.7488. Although this is only a minor difference, it proves that power heuristic indeed has lower variance.

4.7 Bi-directional Path Tracing

An important fact of the sampling methods we have discussed so far is that the variance level depends on both geometry of the emissive surfaces or lights and the local geometry of surrounding surfaces. For brute path tracing, the rate of hitting the background (out of the scene) will be much larger than hitting the light if the summed area of lights is too small or lights are almost locally occluded, which results in only a few of terminated rays would carry the color of emission, causing large variance. For importance sampling or multiple importance sampling, shadow rays must be shot to the light. Expected variance level can

only be guaranteed when the chance of misses is of the same order of magnitude as that of chance of hits; otherwise, the variance level will degenerate to that of brute path tracing. Thankfully, this problem can be solved by also shooting a ray from the light and "connect" the end vertices of the eye path and light path to calculate the contribution, which is called bi-directional path tracing (Lafortune & Willems, 1993).

The core problem in bi-directional path tracing is the "connect" process. Since the contribution of incoming radiance is sampled as a point (end vertex of light path) in the area domain, we must convert that to the solid angle domain to obtain the result. The $\int_{\Omega} f(\boldsymbol{x}, \omega_{in} \rightarrow \omega_{out}) L_{in}(\boldsymbol{x}, \omega_{in}) cos\theta_{in} d\omega_{in}$ term in the rendering equation can be converted to an equivalent $\int_{\mathrm{A}} f(\boldsymbol{x}, \boldsymbol{x}'' \rightarrow \boldsymbol{x}, \boldsymbol{x} \rightarrow \boldsymbol{x}') V(\boldsymbol{x} \leftrightarrow \boldsymbol{x}') G(\boldsymbol{x} \leftrightarrow \boldsymbol{x}') dA$ in the domain of all surface areas, where $V(\boldsymbol{x} \leftrightarrow \boldsymbol{x}')$ is the visibility factor which equals to 1 if the connection path is not occluded and 0 otherwise and $G(\boldsymbol{x} \leftrightarrow \boldsymbol{x}') = \frac{|\widehat{\boldsymbol{n}} \cdot \widehat{\boldsymbol{x}-\boldsymbol{x}'}||\widehat{\boldsymbol{n}'} \cdot \widehat{\boldsymbol{x}-\boldsymbol{x}'}|}{||\boldsymbol{x}-\boldsymbol{x}'||^2}$ is the geometry term.

Importantly, we not only connect the terminated end points of light path and eye path, but the intermediate path ends as well. However, since each connection involves a ray-triangle intersection test, performance will be greatly affected - for path lengths of O(n), there will be $O(n^2)$ intersection tests. In some situation like perfect mirror reflection, we can exclude the contribution of light path since it has a zero probability as a way to save computation time. It is worth mentioning that for all combinations with a specific path length, the contribution of each should be divided by the total number of combinations to maintain energy conservation. For any combined path length of n, there are n + 1 ways of combination of eye path and light path if we want to include the contribution of all kinds of combination. It is also possible to apply importance sampling here as a specific path combination can be weighted by the p.d.f. in all path combinations of the same length. However, that requires additional space to store the intermediate results and may not be good for GPU performance. Next event estimation can also be applied here, so that direct lighting component exists in all combined path length less than or equal to (maximum eye path length + 1), for which the denominator of contribution should be incremented by 1 to include this factor.

In order to test the effect of bi-directional path tracing, we simply inverted the light in the original Cornell Box scene such that light faces the ceiling and the light can only bounce to other surfaces via the small crevices on the rim of the light. The sample image in Figure 8 shows that both rendered in 100 frames, bi-directional path remarkably reduces the noise level comparing to that when only using next event estimation, totally worth for the reduction of frame rate to 50%.

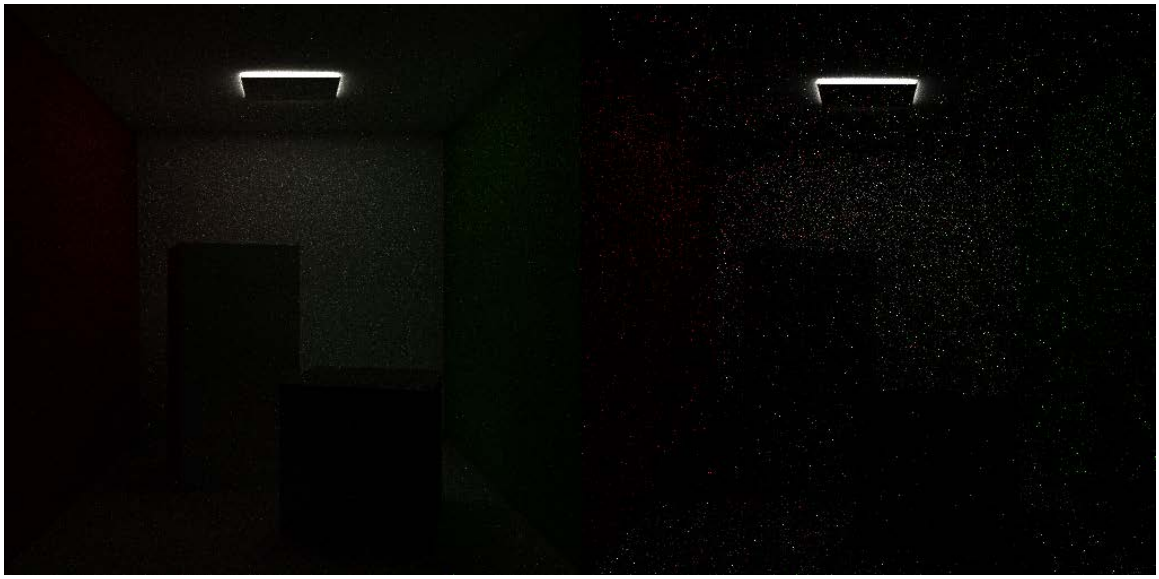


Figure 8 Left: Bi-directional path tracing Right: Uni-directional (eye) path tracing

4.8 Metropolis Light Transport

Using multiple importance sampling and bi-directional path tracing, it is still difficult to maintain low variance in integration estimation for solving problems like bright indirect light, caustics caused by reflected light from caustics and light coming from long, narrow, and tortuous corridors. The key problem of previous sampling methods is that they only consider local importance (light or surface BSDF) rather than the importance of the whole path. In terms of global importance sampling, the original Monte Carlo method is still a brute force solution. To our rescue, there is a rendering algorithm called Metropolis Light Transport (MLT) adapted from the Metropolis–Hastings sampling algorithm based on the Markov Chain Monte Carlo (MCMC) method (Veach, 1997). It has the nice feature that the probability of a path being sampled corresponds to its contribution in the global integral of the radiance toward camera and such paths can be locally explored by designing some mutation strategy. Basically, it proposes new perturbation or mutation to current path in

every iteration and accepts the proposal with a probability $a(x'|x) = \min(1, \frac{f(x')T(x\,|x')}{f(x)T(x'|x\,)})$, where the $f(x')$ and $f(x)$ are the radiance values and $T(x\,|x')$ and $T(x'|x)$ are tentative transition functions which indicate the probability density of transforming from a state to another state in the designed mutation strategy. In general cases, $T(x\,|x')$ and $T(x'|x)$ are not equal. For example, given the specific task of sampling caustics, we can define a mutation strategy that only moves the path vertices at the specular surface. As a return, such mutation can have a transitional probability corresponding with local p.d.f. – if such point $p \in x$ in a specular surface contributes more highlight than another $p' \in x'$ in the same surface as its BSDF suggests, the probability of moving from x' to x is greater than moving from x to x', giving $T(x'|x) > T(x\,|x')$. However, different mutation strategies need to be designed in different kinds of task in order to achieve the highest rendering quality. For general kinds of task, we can ignore local p.d.f. in mutation while still maintaining a good performance. A way of doing so is to store and mutate only the random numbers for every samples generated in the path (selecting camera ray, choosing next ray direction, picking points on area light, determining Russian roulette value, etc.), which will function as "global-local" perturbations on current path, as suggested by Pharr & Humphreys (2011).

Another issue for MLT is ergodicity (Veach, 1997). The MCMC process must traverse the whole path space without getting stuck at some subspaces, which turns out to have a solution of setting a probability for large (global) mutation. Each iteration will test a random number against the threshold. If, for example, the random number is lower than the threshold 0.25, it means on average there is 1 large mutation and 3 small (local) mutations out of 4 mutations. The local mutations are sampled by an exponential distribution (Veach, 1997), implying much larger chance of less movement from the original place while allowing moderately large local mutations. The global mutations are sampled uniformly across the [0,1] interval.

Still another problem of MLT is that in order to choose a probability density function p corresponding to the radiance contribution of the path, which is a scalar, we must find a way to map the 3D radiance value to 1D space to determine the acceptance probability. A

reasonable way of doing this is using the Y value in XYZ color space which reflects the intensity perceived by human eye. A simple conversion formula is given as Y = 0.212671R + 0.715160G + 0.072169B. Note that no matter what mapping formula is chosen, the result is still unbiased. Choosing a mapping closer to eye perception curve allows faster convergence and better visual appearance in same number of iterations.

Nevertheless, another important issue of MLT is the start-up bias (Veach, 1997). Considering the estimation function $m_j = E[\frac{1}{N}\sum_{i=1}^{N}\frac{w(X_i)f(X_i)}{p(X_i)}]$, where f is the radiance, p is the mapped intensity and w is the lens filter function. The Metropolis–Hastings algorithm guarantees that the sampling probability $p = \frac{f}{\int_S f dX}$ in equilibrium, giving the minimal variance. However, we have no way to sample in such p before equilibrium is reached but gradually converge to the correct p.d.f, even though we use $\frac{f}{\int_S f dX}$ as the denominator which is actually not the real p.d.f. for current sample. This causes incorrect color in first few samples, known as start-up bias. Depending on the requirement, if the rendering task is not time-restrictive or not aimed for dynamic scene, we can just discard the first few samples until the result become reasonable. Using a smaller large mutation rate is also a remedy for start-up bias. However, tradeoffs are that complex local paths become harder to detect and while the global appearance converges quickly, local features like caustics and glossy reflection emerge very slowly. In practice, if the scene contains mostly diffuse surfaces, large mutation rate can be set larger. On the other hand, if the intricate optical effects are the emphasis of rendering, large mutation rate should be set much smaller. In fact, designing specific mutation strategies with customized transition functions may be better than just using “global-local” mutations.

It turns out that MLT can be trivially mapped to GPU by running independent tasks in each thread, as is implemented by this project. However, such method still has its defects. For decent convergence rate, the number of threads is set to be equal to the number of screen pixels (so that for the average case, every pixel can be shaded in every frame), which implies high space complexity, due to stored random numbers (stored as float) in graphic memory. If the screen width and height are W and H, maximum path length (or combined path length if bi-directional path tracing is used) is k and about 10 samples need to be

generated for each path segment (as in our implementation), there will be 40k*W*H bytes in total for storing the MLT data. With 1920*1080 resolution and a reasonable k = 30, there will be 2.37G of data, exceeding size of the graphic memory for many low to mid end graphic cards nowadays. To solve this issue, some data compression can be done and it is also possible to let the threads collaborate in a more efficient way, which means running less number of tasks while keeping same or lower variance. However, we will not study this topic in this project due to the effectiveness of existing performance and existence of other important issues.

Last but not least, the estimation of global radiance $\bar{r} = || \int_S f dX ||$ (||x|| indicates a measure of the magnitude of intensity of the RGB color) at the beginning will affect the overall luminance of the final result. From the formula $m_j = E[\frac{1}{N}\sum_{i=1}^{N} \frac{w(X_i)f(X_i)}{p(X_i)}]$, $p(X_i)$ is chosen to be $\frac{||f(X_i)||}{\bar{r}}$, yielding $\frac{\bar{r}w(X_i)f(X_i)}{||f(X_i)||}$ as the radiance contribution from one sample, where $\bar{r}$ functions as a scaling factor for all samples. As a result, in the case where the rendered result is required to be physically authentic, more sample should be taken to estimate the global radiance, although it causes a considerable overhead.

To illustrate the advantage of MLT, some sample image from tests are shown in Figure 9. In both cases, bi-directional path tracing with multiple importance sampled direct lighting is used; the first row shows the result in frame 1, 30, 100 for normal Monte Carlo (MC) sampling whereas the second row shows the results at same frames for MLT. Reader will notice difference of the ways of the two estimators accumulate color. While maintaining a low noise level from the initial frame, MLT estimator exhibits start-up bias as shown by the dim color of the part of the ceiling directly illuminated by the light. In frame 100, the MLT estimator almost reaches the equilibrium as observed in comparison with the MC estimator with respect to color of directly illuminated part of the ceiling. It is worth mentioning that such level of variance is can only be achieved in frame number > 1000 by MC estimator while the MLT estimator is only slightly slower than MC estimator, cause of which mainly attributes to the atomic memory write as each thread runs an independent MLT task which can write color to all pixels on the screen.

Figure 9 Upper: Monte Carlo Lower: Metropolis (Both use bi-directional path tracing)

Chapter 5 Rendering Effects

Before going to the discussion of SIMT optimization, we present this chapter to briefly introduce the rendering effects supported by the path tracer, the importance of which comes from the fact that it is the direct application of the sampling methods discussed before.

5.1 Surface-to-surface Reflection/Refraction

As a guarantee of its practical capability, our path tracer simulates the optical effects of all kinds of surface-to-surface reflection or refraction, not including the cases like polarized light or fluorescence which are rare in practice. For diffuse reflection, the Lambertian model adjusted by Ashikhmin and Shirley formula is used (Section 4.5) while Cook-Torrance microfacet model is responsible for specular or glossy reflection. Our path tracer also supports anisotropic material standardized by Wald model (Ward, 1992), which only modifies the Beckmann distribution factor in Cook-Torrance model (Cook & Torrance, 1982),

$$D_{iso} = \frac{e^{\frac{-tan^2\theta}{\alpha^2}}}{\pi\alpha^2 cos^4\theta}\ ;\quad D_{aniso} = \frac{e^{-tan^2\theta(\frac{cos^2\phi}{{\alpha_x}^2}+\frac{sin^2\phi}{{\alpha_y}^2})}}{\pi\alpha_x\alpha_y cos^4\theta},$$

where α_x and α_y correspond to "roughness" of the material in x and y direction w.r.t. tangent space. Taking azimuth angle ϕ as argument, it is easy to see that when $\phi = 0\ or\ \pi$ the distribution is completely determined by α_x and when $\phi = \frac{\pi}{2}\ or\ \frac{3\pi}{2}$ the distribution is totally decided by α_y.

For importance sampling the Wald BRDF, two uniform unit random variables ξ_1 and ξ_2 are generated and it is easy to solve the equations for azimuth angle ϕ and altitude angle θ:

$$\phi = \arctan\left(\frac{a_y}{a_x}\tan(2\pi\xi_1)\right),\qquad \theta = \arctan\left(\sqrt{\frac{-log\xi_2}{\frac{cos^2\phi}{{\alpha_x}^2}+\frac{sin^2\phi}{{\alpha_y}^2}}}\right)$$

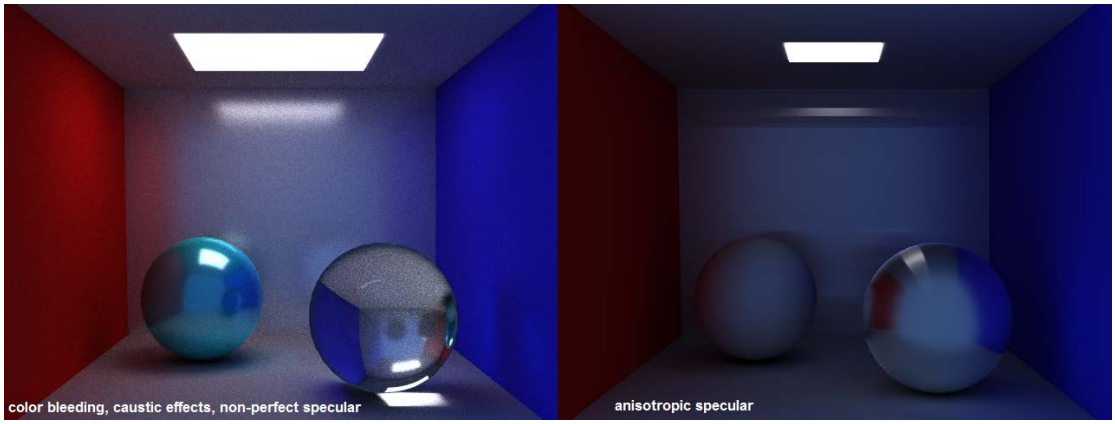


Figure 10 Isotropic & anisotropic specular

For surface-to-surface refraction, the Cook-Torrance microfacet model can be modified by recalculating the Jacobian matrix for the transform between half-vector **H** and outgoing vector **L (**Walter, Marschner, Li, Torrance, 2007**)**, yielding $f_T = \frac{(1-F)GDJ\,|V\cdot H||L\cdot H|}{|V\cdot N||L\cdot N|}$, where D is the microfacet distribution function (Beckmann in our implementation, G is the shadowing term (a numerical approximation to Smith shadowing function in our implementation, the roughness coefficient α inside whom is substituted by $1/\left(\frac{cos^2\phi}{{\alpha_x}^2} + \frac{sin^2\phi}{{\alpha_y}^2}\right)$ for anisotropic material) and $J = \frac{{\eta_o}^2|L\cdot H|}{(\eta_i|V\cdot N|+\eta_o|L\cdot N|)^2}$, where η_i, η_o are the index of refraction of the two media, is the absolute value of the Jacobian matrix. The half-vector **H** in refraction indicates the normal of the sampled microfacet, which can be obtained by calculating $-(V\widehat{+\eta_o}L)$ if BSDF needs to be determined from arbitrary incoming and outgoing radiance so as in the case of multiple importance sampling.

5.2 Volume Rendering

The rendering techniques so far are based on the assumption that spaces between surfaces are vacuum, which is only an effective simplification in ordinary cases with clear air. For phenomena like fog, smog, smoke, obvious scatter, absorption, and emission can happen between surfaces, which affect the radiance towards viewer. In the presence of such participating media, an integro-differential equation of transfer (Chandrasekhar, 1960) shows the directional radiance gradient of a point in participating media to model the change of radiance in space.

$$\frac{\partial}{\partial t}L_o(x,\omega) = -\sigma_t(x,\omega)L_i(x,-\omega) + L_{ve}(x,\omega) + \sigma_s(x,\omega)\int_S p(x,-\omega' \to \omega)L_i(x,\omega')d\omega',$$

where x is the point in space and ω is the direction in interest with t being the measure of displacement along the direction. σ_t is the attenuation coefficient accounting for both absorption and out-scattering, while σ_s is the scattering coefficient controlling the magnitude of in-scattering, which has the phase function p to define the probability density of in-scattering from each direction. L_{ve} and L_i stand for media emission and incoming radiance, respectively. For isotropic media, p has a constant value of $\frac{1}{4\pi}$, which is intuitive as the integral of differential angles in the sphere gives 4π. For general media, there is a phase function developed by Henyey and Greenstein (1941) which provides a simple asymmetry parameter g ranging from -1 to 1 to control the "polarity" of the participating media.

In practice, the integro-differential equation is solved by decomposing different parts, calculating their values separately before using ray marching to accumulate the values in each sample point on the incoming ray. These sample points are treated as differential segments with constant coefficients. Indeed, such numerical integration can be estimated by Monte Carlo sampling. The parameter t of ray can naturally be used as the measure of displacement along the ray direction. Depending on the targeted convergence rate or frame rate, the step size can be set larger or smaller. Normal Monte Carlo sampling would randomly pick a point in each segment for radiance estimation. However, for estimating the light transfer integral which is only one dimensional and often has a smooth function, standard numerical integration may have an edge over the Monte Carlo method. By using a stratified pattern (Pauly, 1999) which assigns same offset for each segment and randomizes the offset for a new ray, it can be shown to have a lower variance than Monte Carlo. The p.d.f. for such samples is also very simple, which is a constant equal to the step size Δt.

In the simplest case of volume rendering, where the participating media has homogenous attributes, both emission and attenuation factors can be directly evaluated by $e^{-\sigma s}$ (where s is the length of the ray segment of interest), which is the direct result of solving the

differential equation $\frac{\partial}{\partial t} L_o(x,\omega) = -\sigma_t(x,\omega) L_i(x,-\omega)$, also known as Beer's law. If the distribution of media density or other properties has an analytical solution for such differential equation, the analytical solution (if it is an elementary function) can be directly evaluated without using sampling techniques, which is used where an analytical solution is impossible, unknown, or too complex, or in the case where the distribution is a customized discrete data set.

As mentioned before, the attenuation term and the augmentation term can be treated separately in computation, which maps well to the accumulation of mask and intermediate color in our implementation, expressed by

$$L_i(x,\omega) = T(x_0 \to x) L_o(x_0,-\omega) + \int_0^T T(x' \to x) S(x' \to -\omega) dt,$$

where $T(x_0 \to x)$ is multiplied to the mask and $\int_0^T T(x' \to x) S(x' \to -\omega) dt$ is estimated with samples and added to the immediate color. Transmittance coefficient T can either be analytically evaluated or estimated with samples as mentioned in previous paragraph.

The sampling estimation of the augmentation term $\int_0^T T(x' \to x) S(x' \to -\omega) dt$ can use the same stratified pattern mentioned before to reduce the variance. However, inside $S(x' \to -\omega) = L_{ve}(x,\omega) + \sigma_s(x,\omega) \int_S p(x,-\omega' \to \omega) L_i(x,\omega') d\omega'$, there is another integral which accounts for in-scattering from all directions, the estimation of which is another non-trivial task. For simplicity, we only consider single scattering from direct lighting. A light sample can be taken as in next event estimation and a shadow ray is shot from the current ray segment's sample point to the light to detect visibility. Note that this method neglects the contribution from indirect lighting to the in-scattering, which is often too weak to affect the rendering equality.

It is worth mentioning that it is also possible to use metropolis light transport for sampling the in-scattering. A random number can be stored here for mutation in every frame so that directions with large contribution can be easily discovered and focused on, which is especially suitable for very anisotropic participating media.

Two sample images are shown in Figure 11 to exhibit the visual effect of volume rendering. The first image shows strong scattering and absorption of light in a room with dense homogenous smog. The smog has a Henyey-Greenstein asymmetry parameter of 0.7, indicating that incident lights are primary scattered forward, as can be seen in the narrow shape of the illuminated cone under the area light. The second image illustrates fog with white emission and exponentially attenuated density in vertical direction, which is the miniature of the atmosphere in a box.

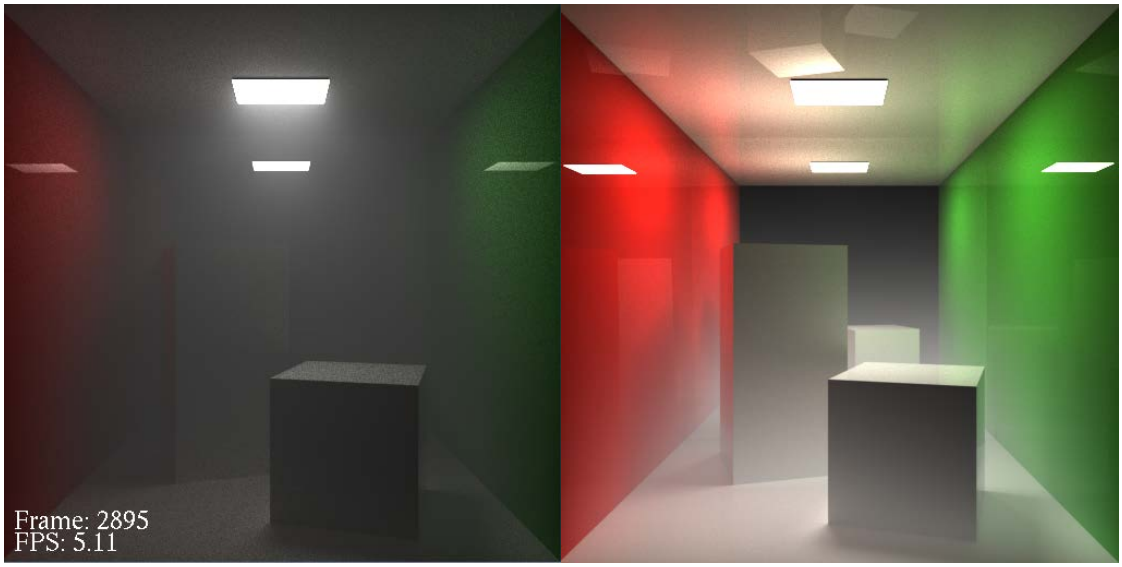


Figure 11 Left: Homogenous smog with strong absorption and forward scattering Right: Fog with exponential density

5.3 Subsurface Scattering

Scattering and absorption can happen inside objects as well. The reason for using BSDF to estimate the radiance from material is that many material can be categorized into metallic (which reflects most of the energy at surface) or dielectric which are either too opaque or too transparent to exhibit any obvious scattering effect. For material with an albedo high enough to be considered as non-transparent and not enough to be considered as opaque like jade, milk and skin, the effect of scattering inside cannot be ignored, for which BSDF cannot give sufficient approximation of the surface radiance. Instead, BSSRDF (bi-directional subsurface scattering reflectance distribution function) is used to include the contribution to the outgoing radiance of the point in interest from incoming radiance on other surface points. A 6-dimensional function

$S(x_i, \omega_i, x_o, \omega_o)$ (x_o is known, the other 3 variables are all 2D) can be used to describe the sum of all radiance scattered from x_i to x_o in all possible paths. Again, to calculate the outgoing radiance of the point in interest, one must integrate the contribution from points all over the object surface, which can be estimated by randomly sampling a surface point as a Monte Carlo process. However, the BSSRDF itself is largely unknown due to the complexity of the multiple scattering problem. Also, points in the surrounding may contribute most of the radiance which implies that indiscriminately choosing a surface point has an intolerably low convergence rate.

To provide a practical approximation of general subsurface scattering, Jensen et al. introduced the dipole diffusion model (2001), which decomposes the BSSRDF into a diffusion term and a single scattering term as a simplification. Observing that the radiance distribution becomes nearly isotropic after thousands of scatterings in material with very high albedo like milk, they proposed a diffusion model that transforms the incoming ray into a dipole source and uses the radial diffusion profile of the material to compute the outgoing radiance. The diffusion term has an exponential falloff with respect to the distance from the incidence point, which provides an effective p.d.f. for importance sampling. Note that the key idea of this model is to interpolate between 2 extreme cases - pure single scattering and pure diffusion – for general material, which turns out to be an insufficient approximation when highly physically authentic pictures are required.

In our implementation, we only consider the contribution of the single scattering term as a demonstration of the idea of subsurface scattering. The program can be easily extended with an additional module for the diffusion term estimation. Since single scattering only happens when the refraction rays of ω_i and ω_o meet inside the material, BSSRDF is not directly evaluated by taking a surface sample. Instead, after the ray intersects the surface with BSSRDF component, a random distance is generated by $\frac{-\log(\xi)}{\sigma'_s}$, where ξ is the unit uniform random variable and $\sigma'_s = (1-g)\sigma_s$ is the reduced scattering coefficient corresponding to the tendency of forward scattering, followed by moving the intersection point by such distance in the ray direction to become the point of single scattering and using the phase function p to importance sample the direction of scattering as the direction of the next ray. Note that this method is only suitable for rendering translucent objects with

low to moderate albedo like jade. Objects with high albedo still require at least the dipole model to render a reasonable appearance.

A pair of sample images are shown in Figure 12 to show the Stanford bunny with a deep jade color rendered as translucent and opaque material. The result of the translucent material shows the effect of subsurface single scattering. Note that thin parts of the object like the ear has a more acute response to the change of albedo.

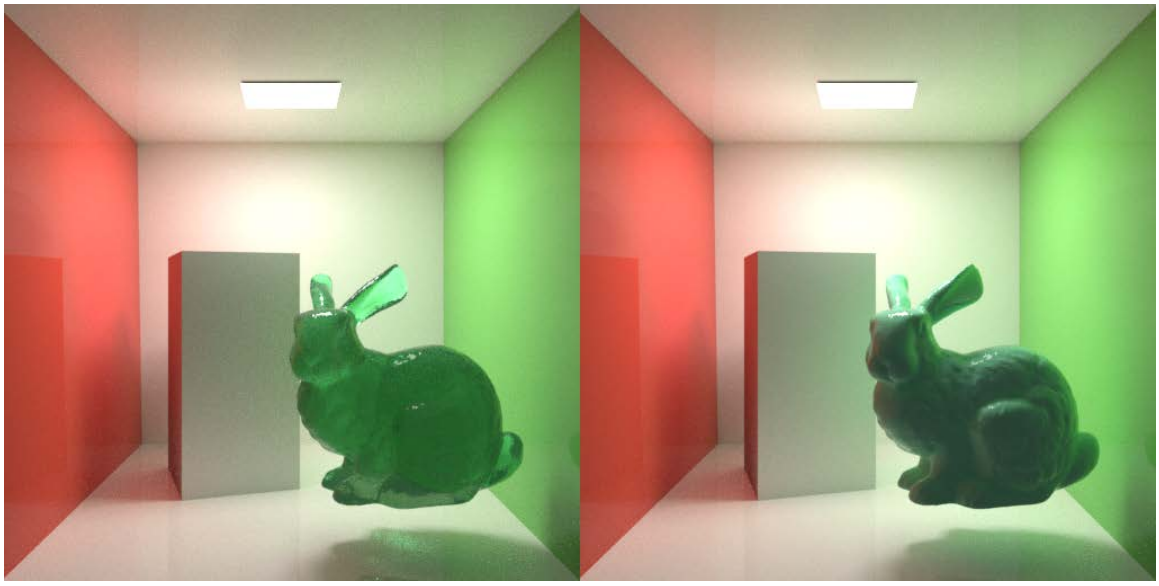


Figure 12 Left: Bunny with subsurface scattering Right: Bunny without subsurface scattering

5.4 Environment Map and Material Texture

As mentioned in the introduction, the heavy usage of textures can often be found in rasterization based graphics used in most of the mainstream video games. PBR workflow, as the de-facto industrial standard, uses a variety of textures to describe the varying material attributes across the texture space. Apart from the basic diffuse color texture or albedo map, a roughness map is used to define the shape of the distribution of normal in the microfacet model, such that the roughness value of 0 indicates a perfectly smooth surface point which only reflects at the mirroring direction and the roughness value of 1 indicates a surface point with almost equal amount of reflection in every direction which makes it no longer look glossy. Similarly, a metalness map is used to define the level of similarity to metal of each texel. A metalness value of 1 defines a surface point that only has specular reflection with some absorption of specific wavelengths which simulates the behavior of metal, while

a metalness value of 0 defines a completely dielectric surface point that follows the Fresnel equation with real index of refraction. More commonly, a uniform white specular color is defined for every opaque object to substitute the evaluation of R_0 in the Fresnel equation when index of refraction is not defined. Furthermore, a normal map is used to simulate microscopic geometry for material with a complex surface texture and ambient occlusion maps are used to compensate the corresponding effect in the lack of global illumination in rasterization based graphics. For still objects, light maps which baked the color bleeding or shadow (requiring still light model) can be used in surrounding surfaces like walls or floors to give a more realistic feeling of the rendering. Since it is generally infeasible to generate real-time samples in rasterization based graphics, usually an irradiance map and a pre-filtered mipmap are precomputed from the environment map of the scene to simulate diffuse and glossy reflection of indirect lighting respectively.

However, for path tracing, we only need the maps related to material attributes in texture space, i.e. albedo map, metalness map, and roughness map, because we are using a global illumination algorithm. We may also want to use environment maps as image-based lights – lighting from the outer environment like sky and sunlight are usually hard to be modeled as solid objects. Also, if the surrounding environment is sufficiently far, using environment map can avoid the ponderous task of modeling all objects from the border of the bounding environment to the point being shaded and accumulating all possible indirect lightings between any two objects. One can simulate the intricate indirect lighting effect by sampling the texel in the reflected ray direction if nothing in the local setting (the models in interest) is hit, treating the outer environment as an infinitely far sphere so that all rays can be seen as being shot from the center of the sphere. The environment map can either be stored as a cubemap or a spherical projection map. In our implementation, we use spherical projection maps due to easier computation and less chance of artefact.

Two sample images are shown in Figure 13 to demonstrate the effect of environment map as image based lighting and the material texture as surface attribute control. The second image simply provides albedo, roughness and metalness map to transform the diffuse back wall of the original Cornell box to a realistic scratched metal.

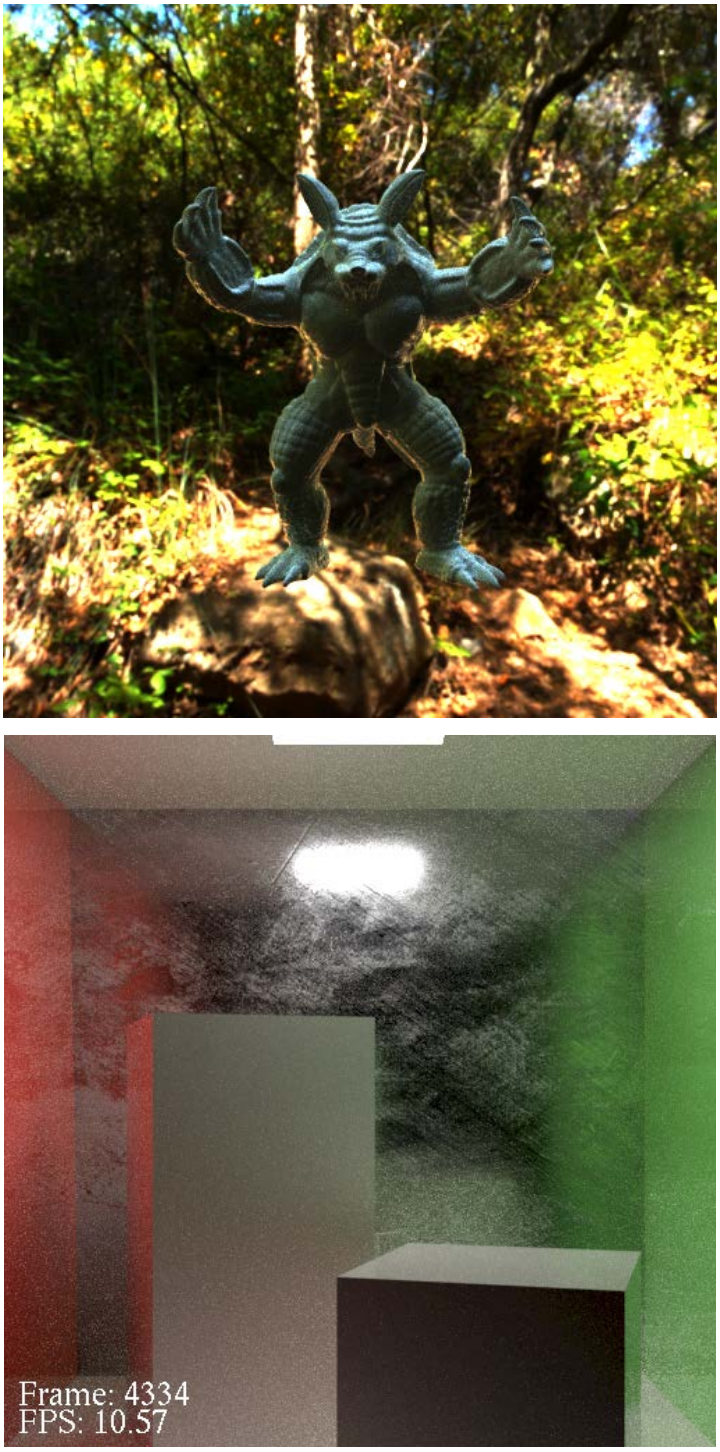


Figure 13 Upper: Armadillo under environment lighting Lower: Cornell Box with scratched metal wall

Chapter 6 SIMD Optimization

With each thread rendering a screen pixel, the problem of path tracing can be solved in an embarrassingly parallel way, without the need of inter-thread communication. However, it is hard to exploit the full capability of single-instruction-multiple-data (SIMD). There is very little locality in the memory access pattern due to generally inconsistent scene geometry, which means almost all scene data need to be stored in global memory or texture memory. Even though the memory access pattern may be relatively congruent in the first bounce, it can be as divergent as possible in the consequent bounces, implying low effective memory bandwidth. Moreover, we have thread divergence in the code everywhere. Although some if-else branches can be merged to a single branch, some are harder or impossible to do so. For example, the branches of different BSDF types contain different sampling functions and different material parameters, which is impossible to merge as one. Special methods need to be designed to alleviate this issue. In addition, the Russian roulette terminates thread with some probability, causing diminishing useful operations and warp occupancy as the iteration number increases.

The following sections will introduce three types of optimizations based on CUDA architecture – data structure rearrangement, thread divergence reduction and thread compaction used in our path tracer to increase the SIMD efficiency and reduce the overall rendering time. The necessity of most of these optimizations comes from the real-time rendering requirement, without the possibility to design fixed number of samples for each rendering branch. After that, two sections will be dedicated to discussion of optimizations on specific components - a ray-triangle algorithm better for SIMD performance will be introduced and we will propose a method for parallel construction of kd-tree on GPU.

6.1 Data Structure Rearrangement

Improper memory access pattern is often the bottleneck of a SIMD program. Therefore, it is our first priority to rearrange the data structure before other components are inspected for optimization.

First, “flattening” the data structure to continuous memory spaces is a key method to improve memory coalescing and reduce memory access. Using kd-tree as SAS, a traditional CPU path tracer stores a tree structure with a deep hierarchy of pointers (Figure 14).

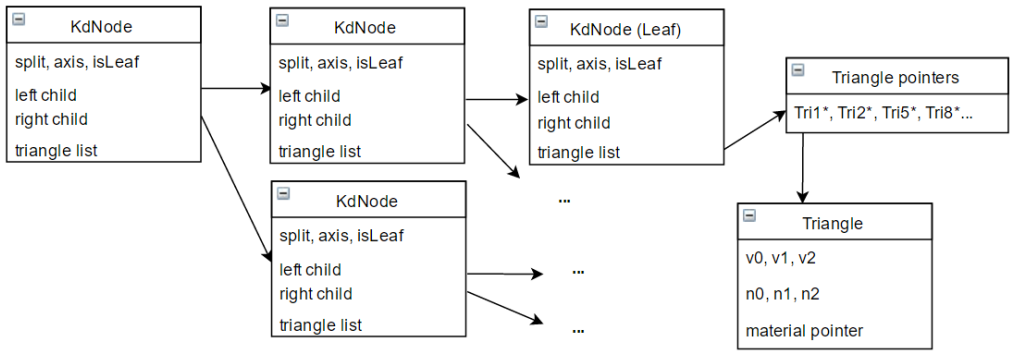


Figure 14. The commonly-used tree structure of kd-tree

Undoubtedly, this structure is unsuitable for GPU. The dynamic memory allocation can give very bad memory coalescing, seriously limiting the effective memory bandwidth in path tracing. One can easily flatten the kd-nodes to an array with the child pointers replaced by child indices, giving an array-of-structures (AoS). However, this is far from the optimum. Instead of keeping a separate triangle indices list for every node, we can store the pointers to triangles continuously in an array and keep only the array offset in node structure. This in large chance gives either coalesced memory access or better cache use because unlike triangles, kd-nodes have a better locality - if we use serial recursion in kd-tree construction, indices of nodes near the bottom of the tree with a near common ancestor will be very near to each other. Similarly, the triangle data can also be stored in an array, with pointers in the triangle list array substituted by indices.

Second, compression of the data structure is another aspect we need to concern about to improve memory efficiency. Notice that in the above kd-node structure, we have some variables that can be represented using few bits – axis (0, 1 or 2 for x, y or z), isLeaf (0 or 1) and the number of triangles (a leaf only contains a few triangles) if we want to only keep offsets in the global triangle list. Notice that the isLeaf predicate can be merged with the axis predicate so that an axis value of 3 indicates a leaf. Rather than using separate variables to store them, one can compress them to one variable. In my path tracer, axis (which

contains isLeaf), number of child triangles (in case of leaf), and left child index are compressed into one 32-bit integer with a 2|6|24 partition using bit operations, which helps to compress a kd-node from 25 bytes to 16 bytes, where 3 words are reserved for split position, right child index and triangle array offset. By compressing left child index from 32 bits to 24 bits, it limits the number of kd-nodes to 16,777,216, which is enough for most cases. The 16-byte compression not only reduces space complexity, but provides memory alignment, improving efficiency of memory access.

Third, SoA can be used in place of AoS when spatial locality is high for neighboring threads or statements. As mentioned before, path tracing does not have a consistent locality for each procedure. Thus, a mixture of SoA and AoS can be used to find a balance between fewer memory accesses and more coalesced memory accesses that can optimize the overall speed. The catenated triangle indices array is an example. In addition, some triangle data (precomputed transformation coefficients, to be introduced later) can be extracted to separate SoA to achieve better cache use when iterating through all triangles in a leaf as triangle indices in a leaf are usually closed to each other. In CUDA architecture, a 128-byte cache line is fetched for each global memory access (NVIDIA, 2015). In a loop that visits some continuous elements, if number of attributes is fewer than number of elements, in large probability fewer global memory access will take place as following access of each attribute can be already in the cache.

6.2 Thread Divergence Reduction

Another important factor of SIMD optimization is minimizing thread divergence. The following code snippet (Figure 15) illustrates some strategies used to reduce thread divergence. First, common statements in branches are extracted to minimize number of operations in each branch. Second, in-place swap is used to replace hardcoded-like assignment. Third, if possible, bit operations are used to replace if-else. Branches with different values assigned to the same variable can be substituted by masking and adding the two values.

```
KdNode* first, *second;
if (D[axis] < 0)
{
    first = kdtree->getNode(curnode->right);
    second = kdtree->getNode(curnode->left);
}
else
{
    first = kdtree->getNode(curnode->left);
    second = kdtree->getNode(curnode->right);
}
if (tsplit >= tfar)
{
    curnode = first;
    axis = curnode->axis;
}
else if (tsplit <= tnear)
{
    curnode = second;
    axis = curnode->axis;
}
else
{
    push(second, tfar);
    curnode = first;
    axis = curnode->axis;
    tfar = tsplit;
}
```

before optimization

```
KdNode* curnode;
curnode = props[idx];
...
int first = atil & 0xffffff, second = curnode.right;
if (D[axis] < 0) swap(first, second);

int f = int(floor(tnear - tsplit)) >> 31;
idx = (first & f) + (second & ~f);
if (f < 0 && tsplit < tfar)
{
    push(second, tfar);
    tfar = tsplit;
}
```

after optimization

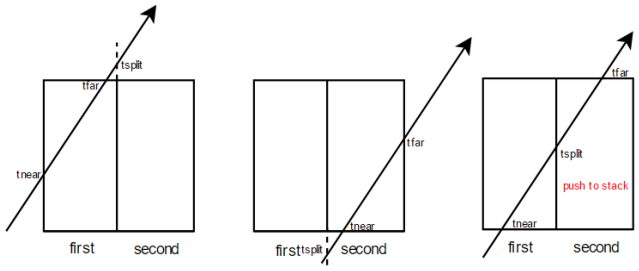


Figure 15. Illustration of an optimization by reducing thread divergence in kd-tree traversal

The snippet also shows another optimization – reduction of global memory access. Rather than storing the pointer to kd-node in stack, it is better to store the index. Otherwise, there will be an extra memory read for every other possibility which will not be executed. The optimization of memory access and thread divergence is often mutual. By reducing memory access, one decreases time taken to execute a divergent branch. By decreasing branch divergence, one reduces possible needs of redundant memory access.

6.3 Thread Compaction

Russian roulette is necessary for transforming the theoretically infinite ray bounces to a sampling process with finite stages, which is terminated by probability. While decreasing the expected number of iterations for each thread in every frame and causing an overall speedup due to early terminated thread blocks, it scatters terminated threads everywhere, giving a low percentage of useful operations across warps (32 threads in a warp are always executed as a whole) and an overall low occupancy (number of active warps / maximum number of active warps) in CUDA, which aggravates as number of iterations increases.

Relating to the set of basic parallel primitives, one naturally finds that stream compaction on the array of threads is very suitable for solving this problem. As illustrated in Figure 16, assuming each warp only contains 4 threads and there is only one block with 4 warps running on GPU for simplification and using green and red colors to represent active and inactive threads, before stream compaction the rate of useful operations is 25% (3 active

threads out of 12 running threads) and after grouping the 3 active threads to a same warp, the percentage of useful operations becomes 75%, equivalently, same amount of work can be done by 1/3 amount of warps, leaving space for other blocks to execute their warps. Also, if the first row is the average case for multiple blocks, the occupancy would be 75% since each block with 4 warps has an inactive warp, implying that less amount of work can be done with same amount of hardware resources. With stream compaction, occupancy is close to 100% in first few iterations, before the time when total number of active threads is not enough to fill up the stream multiprocessor.

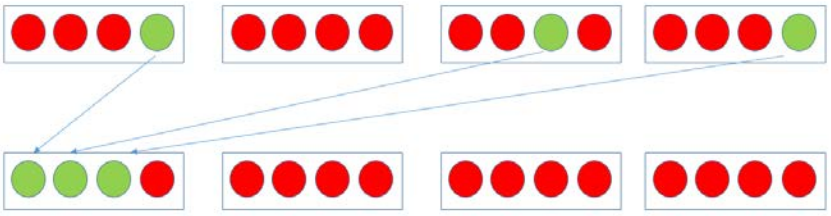


Figure 16 Upper: Before thread compaction Lower: After thread compaction

In order to measure the performance impact of thread compaction, we designed a test comparing frame rate of path tracing with and without thread compaction on our NVIDIA GeForce GTX 960M for maximum trace depth from 1 to 10, and 20, 50, 100. The test scene is the standard Cornell box rendering with next event estimation with 1,048,576 paths traced in each frame.

Max depth	Compact	Normal
1	26.74	48.53
2	19.07	24.23
3	17.74	15.72
4	17.56	12.07
5	17.17	9.73
6	16.05	7.96
7	15.99	7.14
8	16.12	6.04
9	16.23	5.58
10	16.09	5.27
20	15.12	2.84
50	15.28	1.88
100	15.56	1.9

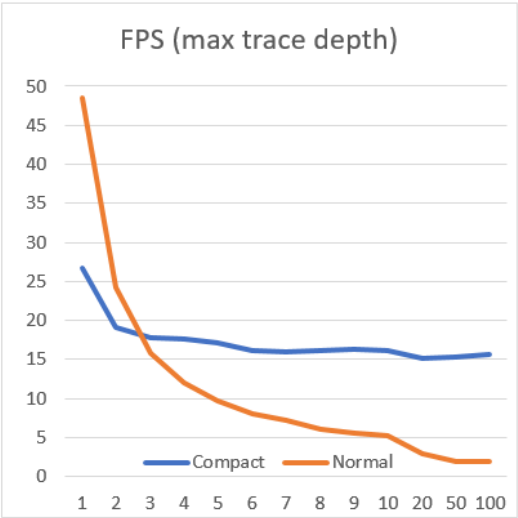


Figure 17 Frame rate as the function of max trace depth, for program with and without thread compaction

As shown by Figure 17, without thread compaction, the frame rate experiences a rapid decline in first 5 increments of max trace depth, after which the declination of frame rate approximates a linear function until the depth when all threads become inactive probably between 20 and 30. With thread compaction, the frame rate starts to surpass the original one in depth 3 with only little falloff for every depth increment and become almost stable after depth 5.

The reason thread compaction causes first two max depths slower is that thread compaction has some overhead of initialization, which cannot be offset by the speedup provided by stream compaction when terminated threads are too few. A struct stores next ray, mask color, pixel position and state of activeness needs to be initialized at the beginning for each thread and retrieved in every bounce. For stream compaction, we also use Thrust library introduced in Chapter 2, which offers a remove_if () function to remove the array elements satisfying the customized predicate. For this task, the customized predicate takes the struct as the argument and checks whether the state of activeness is false to determine elements to discard.

Nevertheless, we can also use stream compaction to do a rearrangement of threads such that threads that will be running the same Fresnel branch in next iteration are grouped together. The number of stream compaction operations will be equal to the number of Fresnel branches (which in our case is 3). By using double buffering, the results of stream compaction can be copied or appended to another array. After generating the resorted array, the indices for the buffers are swapped. In our experiment with a simple scene adapted from the Cornell box with glossy reflection, diffuse reflection and caustics, up to 30% speedup can be achieved from regrouping the threads.

6.4 A More Efficient Ray-triangle Intersection Algorithm

A specific optimization on speed is the adoption of a more efficient intersection algorithm. Ray-triangle intersection can be a performance bottleneck if the math operations are too complex. Schmittler et al. (2004) introduced affine triangle transformation for acceleration of a hardware pipeline. Instead of testing intersection of a fixed ray with varied triangles,

it tests a “unit triangle” against different expressions of the ray in different “unit triangle spaces” from the view of each triangle, which requires an affine transformation.

$$T_{\Delta}(X) = \begin{pmatrix} A_x - C_x & B_x - C_x & N_x - C_x \\ A_y - C_y & B_y - C_y & N_y - C_y \\ A_z - C_z & C_z - C_z & N_z - C_z \end{pmatrix}^{-1} \left(X - \begin{pmatrix} C_x \\ C_y \\ C_z \end{pmatrix} \right)$$

The inverse matrix and the translation term can be computed offline and stored in a 4D float vector for each dimension. Based on extraction of common geometry information, this method reduces the 26 multiplications, 23 additions/subtractions required in the standard ray-triangle intersection algorithm to 20 multiplications and 13 additions/subtractions. By separating the data of precomputed terms from the remaining necessary triangle data (vertex normal and material index) to form a structure of two arrays, an overall speedup of 10-20% can be achieved.

6.5 GPU SAH Kd-tree Construction

We will propose a GPU SAH kd-tree construction method in this section. So far, the CPU construction of SAH kd-tree has a lower bound of O(N log N), which is still too slow for complex scenes with more than 1 million triangles. It takes more than 10 seconds to construct the SAH kd-tree for the 1,087,716-face Happy Buddha model on our Intel i7 6700HQ, which is a serious overhead. Given the immense power of current GPGPU, it is a promising task to adapt the kd-tree construction to a parallel algorithm. A GPU kd-tree construction algorithm was proposed by Zhou et al. (2008), which splits the construction levels into large node stages where median of the node’s refitted bounding box is chosen as the spatial split the and small node stages where SAH is used to select the best split. Although with a high construction speed, the method sacrifices some traversal performance due to the approximated choice of best splits in large node stages. In contrast, we will now propose a full SAH tree construction algorithm on GPU.

First, similar to Wald’s CPU kd-tree construction model (2006), we create an event struct containing the 1D position, type (0 for starting event, 1 for ending event), triangle index (which is actually triangle address since at the beginning the node-triangle association list is same as the triangle list), and a “isFlat” Boolean which marks whether the opposite end

has the same coordinate for every end of bounding boxes of triangles in all 3 dimensions, which are stored in 3 arrays. For each dimension, the event array is sorted by ascending position coordinate while keeping ending events before starting event when the positions are same (we use the same routine as in the Wald's algorithm – subtracting the triangle of ending event from the right side before SAH calculation and adding the triangle of starting event to the left side after the SAH calculation, which can guarantee that triangles with an end lying on the splitting plane can find the correct parent – except for being parallel). Such sort should be a highly efficient parallel sort like the parallel radix sort.

After that, we separate the struct attributes into a SoA (structure of arrays) for better memory access pattern. Also, we need to create an "owner" array of length of number of triangles, which is initialized to zeros as root has an index of 0, to store the index of owner node, since we will be processing the nodes in parallel. So far, we have three position arrays, three type arrays, three triangle address arrays, three isFlat arrays, and one owner array, each of which has the same length of events from all nodes in current construction level. Nevertheless, we also need an array for node-triangle association, which lists the indices of triangles associated with nodes in current level in node-by-node order. Again, this node-triangle association list (which will be called triangle list for short) also needs an owner list, which we call "triOwner", also initialized to zeros.

What still left for initialization are the two dynamic arrays – nodeList for storing all the processed nodes, which are pushed into as groups from the working node array of current construction level, linearly and leafTriList for storing all the triangles in leaves in leaf-by-leaf linear order.

After all initializations are done, we choose a dimension with the largest span in the root's bounding box. Note that the selection of such dimension will be processed in parallel in following iterations, at the moment of creating node structs for all newly spawned children from the current level. The following explanation will treat the current construction level a general level with many nodes other than level 0. The first parallel operation other than sorting we perform is the inclusive segmented scan on the type array, the purpose of which is to count the number of ending events before the current event (or including the current event if it is an ending event) for use in the following calculation of number of triangles on

the left and right side of the splitting plane, alongside with the surface areas of bounding boxes of the potential left child and right child, as is required to calculate the SAH function. In this segmented scan, the owner array is used as a key to separate events from different nodes. It is worth mentioning that for SAH calculation, the offset of the node's events in the event list is stored in the node struct, so that an event is able to know its relative position in its belonging part in the array, which will be used together with the scanned result of number of starting events to the left to derive the number of triangles in the left or right subspace of the splitting plane. For SAH calculation for splitting plane with flat triangle lying on it, we simplified the process by grouping all such flat triangles to the left side, which in most cases has no influence on traversal performance, so that we do not need to deal with the flat case specially in triangle counting. The information of a potential split is stored in a struct containing SAH cost, two child bounding boxes, splitting position, and number of left side and right side triangles. The array of such struct then undergoes a segmented reduction to find the best split (with minimal SAH cost) for each node.

The next step is assigning triangles to each side, which is also the step where we determine whether to turn the interesting node to a leaf. In the assigning function which is launched for every event in current splitting dimension in parallel, we check whether the best cost is greater than the cost of not splitting (which in most cases is proportional to the number of the triangles in the node) or the number of triangles in the node is below a threshold we set. If it is the case, we create a leaf by marking the "axis" attribute in the node struct with 3. For assigning triangles to both children, our key method is to use a bit array of twice the size of the current triangle list and let the threads of current events to assign 1 at the address at the belonging side (or two sides if the triangle belongs to both left and right side), after which the bit array is scanned to obtain the address of the triangle list in next level. Since the events are in sorted order, an event can decide its belonging by comparing the index with the index of the event chosen for best split. If the event is a starting event, and index is smaller than the best index, the event will assign its triangle to the left side; and if the event is an ending event, and the index is greater than the best index, the event will assign its triangle the right side. Notice that because we are launching a thread for each event, a triangle spanning across the splitting plane will be correctly assigned to both side by different threads, without special care. In addition, flat triangles lying on the splitting plane

will be assigned to both sides (where isFlat variable is checked) to avoid the effect of numerical inaccuracy in traversal which can cause artefacts.

Also, a leaf indicator array is assigned by the threads in the triangle assignment function such that the indicator array would have a 1 in the position of triangles that belong to a newly created leaf in the triangle list, which will be scanned to determine the address of the triangle in the leafTriList, similar to how the addresses of triangles in the next level's triangle list are determined, and reduced to obtain the number of triangles in the leafTriList in current level which is used to calculate the new size of the whole leafTriList to be used as next level's offset. Since we also need to know the local offset of the leaf's triangles in the part of current level in leafTriList, we need to do a segmented reduction followed by an exclusive scan on the leaf indicator array before assigning the offset to the leaf's struct.

Before spawning new events for the child nodes, we need to finish the rest of the operations on the triangle list. The triOwner list for the new level can be easily generated by "spawning" a list from the original triOwner list with doubled size by appending the list to itself with the owner index offset by the original number of owners of nodes in the second half and performing a stream compaction using the aforementioned bit array as the key to remove the entries for triangle not belonging to the specific side. A question may be that after the stream compaction, the owner indices are not incremental, which cannot be used for indexing. However, this issue can be easily solved by doing a parallel binary search on the returned key array of the segmented reduction (or counting, more properly) on the constant array of 1 (the returned value array of which is stored as the counts of the triangles in next level's nodes) with the just generated next level's triOwner array itself as the key, whose result is used to replace the array. In a similar way, the triangle list for next level is "spawned" from the original triangle list and compacted by the bit array.

Finally, we explain how the next level's events (type, split position, isFlat and triangle address) are generated. The method is surprisingly simple – after duplicating the event list, we only need to produce a bit array for events by checking the corresponding values in the bit array for triangles, which only requires reading the values in current events' triangle address list as the pointer to the position in the bit array for triangle. The 3 attributes type, split position and isFlat can be spawned by duplicating the original array and perform a

stream compaction with the bit array as the key. The triangle address array itself can spawn the array for next level by duplicating, reading the new addresses in the previously scanned result of the triangle bit array and also doing a stream compaction.

So far, there is only one last array to spawn – the event's owner list in the next level, which can be generated in the same method as the triOwner array uses – "stream compaction – segmented reduction – binary search". Before next iteration begins, node structs for next level are created using data like counts and offsets in the corresponding previous generated arrays and pushed to the final node list as a whole level. The splitting axes for the next level are also chosen in this process by comparing the lengths of the 3 dimensions of the bounding box. If an axis different from current axis is chosen, the 4 event arrays for the 3 dimensions are "rotated" to the desirable place – if 0 stands for the splitting axis and current splitting axis is x, y and z will be stored under index 1 and 2; if next splitting axis is z, the memory will have a "recursive downward" rotation so that z is rotated to 0, x is rotated to 1, y is rotated to 2. Finally, the pointers of all working arrays are swapped with the buffered arrays. The termination condition is that the next level has no nodes.

We also performed a test comparing the speed between Wald's CPU construction and our GPU construction of the same SAH kd-tree (full SAH without triangle clipping) on a computer with Intel i7-4770 processor and NVIDIA GTX 1070 graphics card. The result (Table 2) shows that a sufficiently large model is required for our GPU construction to outperform the CPU counterpart, due to the overhead of memory allocation and transfer. A 5x speedup can be obtained when the model size goes beyond 1M, which indicates that our method can be used for ray tracing large models to greatly reduce the initialization overhead while maintaining the same tree quality.

Model	Face Count	CPU(s)	GPU(s)	Speedup
Cornell	32	0.001	0.046	0.02x
Suzanne	968	0.016	0.095	0.17x
Bunny	69,451	1.442	0.655	2.20x
Dragon	201,031	3.705	1.100	3.37x
Buddha	1,087,716	13.903	2.801	4.96x

Table 2 Speedup of our GPU SAH Kd-tree comparing with Wald's CPU algorithm

Chapter 7 Benchmarking

Benchmarking different path tracing engines is not a trivial task. Different engines have different strengths at different types of rendering tasks. In addition, rendering methods may be different for different engines, thus it is difficult to choose a measure of the performance. If one engine uses Metropolis Light Transport and another engine uses brute force path tracing, one cannot claim that the first engine has a better performance than the second one, just because it has a larger frame rate (or samples per second for offline path tracing). Normally, we compare the performance by convergence rate – in same amount of time, the engine converges more has a better performance – measured by the variance level. A special reminder is that one can only use the variance measure when the engines use same basic sampling method – the only two we introduced before are normal Monte Carlo sampling and Markov Chain Monte Carlo sampling (used only in MLT) – as MLT will always try to find a smallest variance even if the color is incorrect (also known as start-up bias). Alternatively, it seems that one can also compare the absolute difference between the rendered image and the ground truth. However, the BSDF used in different engines are usually slightly different, in case of which the absolute difference is an invalid measure. A practical solution of this issue is to force different engines use the same basic sampling method (in most cases it can be changed in options) and compare the convergence rate. It is important to notice that images generated by different engines must all be tone mapped or non-tone mapped before a variance comparison can be done. Or if it is known that the engines to be compared use the same specific sampling method (like next event estimation or bi-directional path tracing), it is simpler to directly compare the frame rate. However, one should also look at the differences between the rendered image and the ground truth to prevent some low-quality images or artefacts produced by incorrect implementation or the deviation from the industrial standard.

Two scenes are used to benchmark our path tracer against some free mainstream path tracers. The first scene is the default Cornell Box with all Lambertian diffuse surfaces and a diffuse area light, rendered by next event estimation. The real-time path tracing sample program of NVIDIA's Optix ray tracing engine is used to compare with our path tracer

(Figure 18). Since the sample program is open-source and uses the same next event estimation method and both our program and NVIDIA's program are real-time path tracers, we can compare the performance by directly comparing the frame rate of rendering. Table 3 shows the frame rate of rendering of our path tracer and NVIDIA's path tracer in 512x512 resolution with 4 samples taken for each pixel in each frame (Figure 18), on the mid-end NVIDIA GeForce GTX 960M graphics card and the high-end NVIDIA GeForce GTX 1070 graphics card.

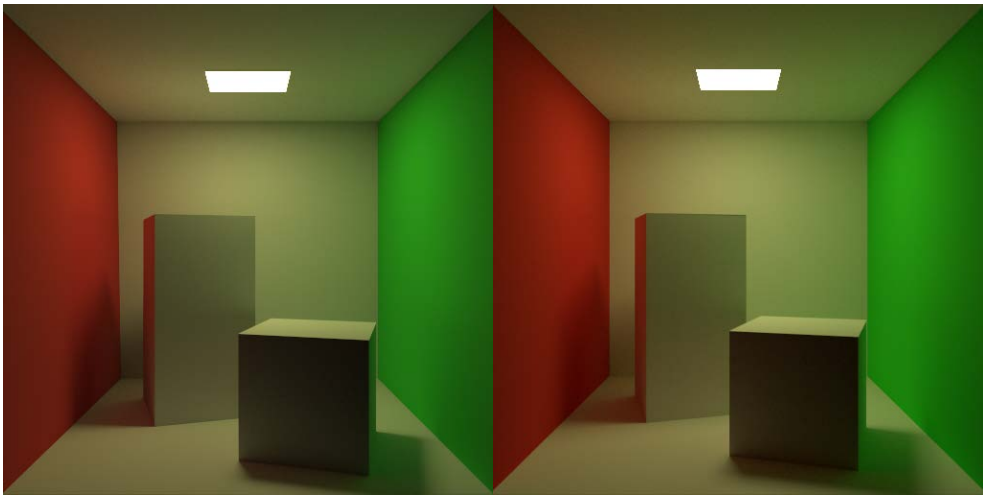


Figure 18 Left: Our Render Right: NVIDIA's Render

	GTX 960M	GTX 1070
NVIDIA's Path Tracer	13.52 fps	30.0 fps
Our Path Tracer	14.02 fps	41.5 fps
Speedup	3.7%	38.3%

Table 3 Speedup of our path tracer on different graphics cards, comparing with NVIDIA's

A reason for our path tracer to gain a larger speedup on high-end graphics card is that high-end graphics cards have larger memory bandwidth, which allows faster memory operations in stream compaction used in our path tracer but not in NVIDIA's path tracer.

The second scene is the BMW M6 car modeled by Fred C. M'ule Jr. (2006), which aims for testing the capability of our path tracer to render models in real application. For comparison, we chose the Cycles Render embedded in the Blender engine. Albeit being an

off-line renderer, it also has a “preview” function to progressively render the result in real-time. Notice that Cycles Render uses a different workflow to blend the material color and may use different BSDF formulae on same material attribute, causing the appearance to be different (the glasses and metal rendered by Cycles is less reflective on same attributes, and the overall tone is different). It is extremely hard to tune the rendering result to the same, but we can still guarantee that the workload on each path tracer is almost the same, as the choice of material component depends on the Fresnel equation.

Since the ways of implementation may also be vastly different, we use the convergence rate in one minute as the measure of the performance. It is important to know that it is invalid to use the variance of all pixel in the picture to compare for convergence. As convergence corresponds to noise level in Monte Carlo sampling, a small region that will be rendered to a uniform color is used for convergence test. For this scene, it is convenient to just choose the upper-left 64x64 pixel to compare for variance, as the wall has a uniform diffuse material which will produce nearly same color under current lighting condition. Also, the rendered result of one hour from our path tracer is used as the ground truth for variance comparison (it is equivalent to use either side’s). For illumination, a 3200x1600 environment map for a forest under sun is used.

The following images in Figure 19 are the grey scale value of the upper-left 64x64 square region from Cycles Render, our path tracer, and the ground truth. By only looking with the eyes, one is difficult to judge which of our result and Cycles’ result has a lower noise level. However, we can numerically analyze the variance by evaluation of the standard deviation of the pixel values. By using OpenCV, the average value and the standard deviation of all grey scale pixels can be easily obtained, which are listed in Table 4.

Figure 19 Left, Middle & Right: Cycles’, our, ground truth's upper left 64x64 pixels in greyscale

	Average	Std. Deviation	Convergence
Theirs	98.42	10.31	16.7%
Ours	98.95	9.41	18.3%
Ground Truth	104.6	1.72	100%

Table 4 Comparing convergence rate of our path tracer and Cycles Render in 1 minute's render time

From the data, we can see that our path tracer does have a slightly better performance than Cycles Render. Although due to time restriction, we are not able to carry out more tests using different scenes and with other mainstream renderers, the complexity of such test scene (700K+ faces, glossy, diffuse, refraction BSDF, environment light) can be a solid proof that our path tracer has at least the same level of performance with current mainstream rendering software. For the reader's interest, we also provide the sample pictures of our and Cycles' rendering results for 1 minute, and our rendering result for 1 hour, which can be found in the appendix.

Chapter 8 Conclusion

8.1 Summary

This thesis focuses on how to improve the solution to the real-time path tracing problem by introducing and discussing possible optimizations in 3 categories – SAS, sampling and SIMD, which are implemented in a program with real-time rendering and interaction capability. While the SIMD optimization bases itself on the parallel computing model in GPGPU and aimed specially for the real-time requirement, the first two categories – SAS and sampling – are not hardware dependent and also used in off-line renderers as they are defined in the domain of a single computing thread. However, it is also possible to improve the models involved in these two categories to achieve better collaboration with the GPGPU model. For SAS, as a common bottleneck of ray tracing processes, SAH based kd-tree and BVH were introduced for being the optimum of their peers in minimizing expected global cost of ray-primitive intersection test and their indispensable functions in different applications, and optimization techniques on such data structure including triangle clipping and short stack traversal for kd-tree and node refitting for dynamic BVH are also discussed with implementation details. In the chapter for sampling, different context-based optimization methods on Monte Carlo algorithm which are all aimed for decrease variance in rendering – importance sampling on BSDF, next event estimation for direct lighting, multiple importance sampling combining the previous two, and bidirectional path tracing for difficult lighting conditions – were introduced. Moreover, Metropolis Light Transport as a modification of the basic Monte Carlo process based on Markov Chain was introduced and some implementation details on GPU were shared. For SIMD optimization, data structure rearrangement, code-level thread divergence reduction, thread compaction as three different types were illustrated with codes and test cases. A more efficient ray-triangle intersection solution which transforms the problem space was cited for its contribution on the performance increase of our program. More importantly, we proposed a new GPU construction algorithm for SAH kd-tree in full details, which turns out to help greatly reducing the initialization overhead for complex model. In addition, the underlying mechanism of rendering effects chosen and supported in our program – surface-to-surface

reflection/refraction, volume rendering, and subsurface scattering were analyzed to clarify possible complications in usage. For most methods we introduced and discussed, test cases on our path tracer were provided to verify the ideas. Finally, we benchmarked our program with the path tracing demo in NVIDIA's Optix engine and a free mainstream path tracer to prove that our program has a large advantage in rendering simple scenes like the Cornell Box by improving the performance by up to 30% and slightly outperforms a free mainstream path tracer for a complex rendering of a car, which means it is at least competitive with most of the mainstream path tracers nowadays in real-time rendering of models with industrial complexity. By analyzing, gathering, testing, and integrating different optimization techniques into a whole process, and choosing the correct rendering methods, we can efficiently produce aesthetically-pleasing, photorealistic results.

8.2 Limitations & Recommendations for Further Work

Given the immense potential of GPGPU, it is possible to see path tracing offering a photorealistic, film standard experience, replacing rasterization-based graphics to be the gaming standard in the future as the hardware performance continues to multiply. However, improvements in algorithm and software structure are also necessary to reduce as much workload as possible to accelerate the coming of such day. This thesis addresses many distinctive issues of real-time path tracing such as large thread divergence and dynamic geometry. However, many problems that may appear in future real-world applications of path tracing have not been considered due to the time limit. One such problem is to efficiently render a large set of animation data which may contain particle system or complex deformation. Another problem is the insufficient optimization of the spatial acceleration structure which is a bottleneck in ray-traced graphics. New algorithms or hardware need to be developed to continuously improve the traversal speed and update or rebuild the SAS with minimal efforts. In addition, better parallelization methods are still required for some algorithms with relatively obscure parallelizability but tremendous serial performance like Metropolis Light Transport, even though many have been developed. Moreover, parsing can be transferred to the GPU to greatly reduce the initialization time of geometrically complex scenes.

Bibliography

Ashikhmin, M., & Shirley, P. (2000). An anisotropic Phong BRDF model. Journal of graphics tools, 5(2), 25-32.

Beason, K. (2007). Smallpt: Global Illumination in 99 lines of C++. Retrieved from http://www.kevinbeason.com/smallpt/

Chandrasekhar, S. (1960). Radiative Transfer. New York: Dover Publications. Originally published by Oxford University Press, 1950.

Chandrasekhar, S. (1960). The stability of non-dissipative Couette flow in hydromagnetics. Proceedings of the National Academy of Sciences, 46(2), 253-257.

Cook, R. L., & Torrance, K. E. (1982). A reflectance model for computer graphics. ACM Transactions on Graphics (TOG), 1(1), 7-24.

Foley, T., & Sugerman, J. (2005, July). KD-tree acceleration structures for a GPU raytracer. In Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware (pp. 15-22). ACM.

Henyey, L. G., & Greenstein, J. L. (1941). Diffuse radiation in the galaxy. The Astrophysical Journal, 93, 70-83.

Jensen, H. W., Marschner, S. R., Levoy, M., & Hanrahan, P. (2001, August). A practical model for subsurface light transport. In Proceedings of the 28th annual conference on Computer graphics and interactive techniques (pp. 511-518). ACM.

Kajiya, J. T. (1986, August). The rendering equation. In ACM Siggraph Computer Graphics (Vol. 20, No. 4, pp. 143-150). ACM.

Kopta, D., Ize, T., Spjut, J., Brunvand, E., Davis, A., & Kensler, A. (2012, March). Fast, effective BVH updates for animated scenes. In Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (pp. 197-204). ACM.

Lafortune, E. P., & Willems, Y. D. (1993). Bi-directional path tracing.

NVIDIA. (2015). Memory Transactions. NVIDIA® Nsight™ Development Platform, Visual Studio Edition 4.7 User Guide. Retrieved from http://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/sourcelevel/memorytransactions.htm

NVIDIA. (2017). CUDA Toolkit Documentation. Retrieved From http://docs.nvidia.com/cuda/thrust/#axzz4dK4GrjBF

Pauly, M. (1999). Robust Monte Carlo Methods for Photorealistic Rendering of Volumetric Effects (Doctoral dissertation, Master's Thesis, Universität Kaiserslautern).

Pharr, M., Jakob, W., & Humphreys, G. (2011). Physically based rendering: From theory to implementation. Second Edition. Morgan Kaufmann.

Santos, A., Teixeira, J. M., Farias, T., Teichrieb, V., & Kelner, J. (2012). Understanding the efficiency of KD-tree ray-traversal techniques over a GPGPU architecture. International Journal of Parallel Programming, 40(3), 331-352.

Schlick, C. (1994, August). An Inexpensive BRDF Model for Physically-based Rendering. In Computer graphics forum (Vol. 13, No. 3, pp. 233-246). Blackwell Science Ltd.

Schmittler, J., Woop, S., Wagner, D., Paul, W. J., & Slusallek, P. (2004, August). Realtime ray tracing of dynamic scenes on an FPGA chip. In Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware (pp. 95-106). ACM.

Veach, E. (1997). Robust monte carlo methods for light transport simulation (Doctoral dissertation, Stanford University).

Vinkler, M., Havran, V., & Bittner, J. (2014, May). Bounding volume hierarchies versus kd-trees on contemporary many-core architectures. In Proceedings of the 30th Spring Conference on Computer Graphics (pp. 29-36). ACM.

Wald, I., & Havran, V. (2006, September). On building fast kd-trees for ray tracing, and on doing that in O (N log N). In Interactive Ray Tracing 2006, IEEE Symposium on (pp. 61-69). IEEE.

Walter, B., Marschner, S. R., Li, H., & Torrance, K. E. (2007, June). Microfacet models for refraction through rough surfaces. In Proceedings of the 18th Eurographics conference on Rendering Techniques (pp. 195-206). Eurographics Association.

Ward, G. J. (1992). Measuring and modeling anisotropic reflection. ACM SIGGRAPH Computer Graphics, 26(2), 265-272.

Zhou, K., Hou, Q., Wang, R., & Guo, B. (2008). Real-time kd-tree construction on graphics hardware. ACM Transactions on Graphics (TOG), 27(5), 126.

Appendix

The following pictures show the result of rendering a BMW M6 car for one minute in Cycles Render, one minute in our path tracer, and one hour in our path tracer, successively. The BMW M6 car model was modeled by Fred C. M'ule Jr in 2006, under CC-Zero (public domain) license, downloaded from http://www.blendswap.com/blends/view/3557.