EFFICIENT RENDERING OF SCENES WITH DYNAMIC LIGHTING USING A PHOTONS QUEUE AND INCREMENTAL UPDATE ALGORITHM

Guangfu Shi

A thesis
in
The Department
of
Computer Science

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Computer Science
Concordia University
Montréal, Québec, Canada

November 2012

Concordia University

School of Graduate Studies

This is to certify that the thesis prepared

By:	**Guangfu Shi**
Entitled:	**Efficient Rendering of Scenes with Dynamic Lighting Using a Photons Queue and Incremental Update Algorithm**

and submitted in partial fulfillment of the requirements for the degree of

Master of Computer Science

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining commitee:

______________________ Chair

______________________ Examiner

______________________ Examiner

______________________ Examiner

______________________ Supervisor

Approved ______________________

Chair of Department or Graduate Program Director

__________ 20 _____ ______________________

Dean

Faculty of Engineering and Computer Science

Abstract

Efficient Rendering of Scenes with Dynamic Lighting Using a Photons Queue and Incremental Update Algorithm

Guangfu Shi

Photon mapping is a popular extension to the classic ray tracing algorithm in the field of realistic image synthesis. Moreover, it benefits from the massive parallelism computational power brought by recent developments in graphics processor hardware and programming models. However rendering the scenes with dynamic lights still greatly limits the performance due to the re-construction at each rendered frame of a kd-tree for the photons. We developed a novel approach based on the idea that storing the photons data along with the kd-tree leaf nodes data and implemented new incremental update scheme to improve the performance for dynamic lighting. The implementation is GPU-based and fully parallelized. A series of benchmarks with the prevalent existing GPU photon mapping technique is carried out to evaluate our approach. Our new technique is shown to be faster when handling scenes with dynamic lights than the existing technique while having the same image quality.

Acknowledgments

First of all, I would like to thank my supervisor, Dr. Thomas Fevens for his advice, encouragement and support throughout my research. I would also like to thank my parents and my girlfriend Chunxiao Ma who are supportive as always.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Due to the recent development of Graphics Process Units (GPUs) hardware architecture and the massive parallel programming model that enables developers to fully exploit the computation power of GPUs, parallel computing using GPU has become more and more popular for developing high performance applications. Realistic image synthesis, especially global illumination which is one of the most computationally complex algorithm, has drawn much research interest focusing on developing new parallelized approaches to unleash the computational power of the GPU.

Photon mapping is one of the most popular global illumination approaches these days. Many of the acceleration techniques developed for photon mapping can achieve very good performance for static scenes. However, dynamic scenes such as a scene in which there are moving lights sources or moving elements in the scene may have a big impact on the performance of renderer. One of the biggest causes of this overhead is that the the dynamic scene requires the reconstruction of the acceleration data structure every frame. The data structure for photon mapping is usually a balanced kd-tree [1] used to speed up the photon search. A parallelized kd-tree construction algorithm will greatly increase the peak memory consumption though it is a pretty fast algorithm, Zhou et al. reported that their GPU-based implementation can achieve above 30 frames per second [30].

In this thesis we would like to present a novel approach based on the traditional photon mapping technique on GPU to be able to handle dynamic light sources. Using this approach we rearrange the photons data to avoid the reconstruction of kd-tree for dynamic scenes with moving light sources. An incremental update algorithm is also employed so that we can achieve the same image quality while requiring less rendering time once an initial number of start-up frames are rendered.

In order to give some proof that our proposed technique is an improvement in terms of speed, memory consumption and image quality, we will implement the prototype of the existing technique from the literature and compare it with the new approach.

1.1 Structure of the thesis

Following this introduction, we will firstly introduce the most important theoretical concepts and frameworks to establish a foundation for the subsequent discussion of global illumination approaches. Then we will present a short survey on various approaches to implement photon mapping and their strengthes and weaknesses when dealing with different phenomena. Afterwards we will highlight GPU-based techniques applied on Monte Carlo ray tracing and photon mapping and discuss some open issues of the current approach for the introduction of our new approach in the following chapter. In chapter three, we presented the new GPU photon mapping approach with detailed information on our current implementation. The results of our experiment and the analysis of the results are presented in chapter four. Finally, in chapter five we come to the conclusions, and discuss directions for future work.

Chapter 2

Background and Related Work

2.1 Radiometry Introduction

Radiometry is the basic terminology to describe light which is crucial to its simulation. First of all, some basic quantities have to be introduced, the related symbols are going to be defined here as well for further use.

2.1.1 Important Quantities

Symbol	Quantity	Unit
Q	Radiant Energy	j
Φ	Radiant flux	W
I	Radiant intensity	Wsr^{-1}
E	Irradiance (incident)	Wm^{-2}
L	Radiance	$Wm^{-2}sr^{-1}$

Table 1: Important Radiometry Quantities

Radiant energy, Q, is the energy of a collection of photons which is the basic quantity in lighting.

Radiant flux , Φ, is the time rate of the flow of radiant energy passing through a surface or region of space. Total emission from a light source is generally described in terms of flux. Figure 2.1 shows the flux emitted from a point light source measured by the total amount of energy passing through an virtual sphere around the light.

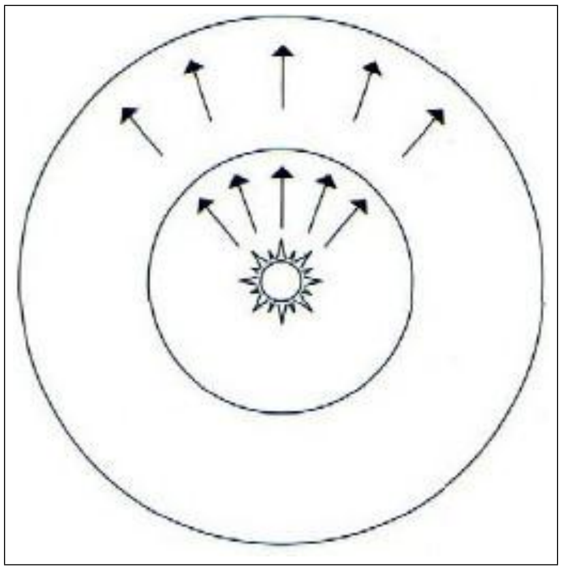


Figure 2.1: Radiant flux from a point light source is passing through the spheres around the light.

Irradiance, E, is the incident (arriving at a surface location) *radiant flux area density*, which is defined as the differential flux per differential area. While *Radiant exitance* denoted by M is area density of flux leaving a surface.

$$E(x) = \frac{d\Phi}{dA} \tag{1}$$

Radiance, L, is the radiant flux per unit solid angle per unit projected area:

$$L(x, \overrightarrow{\omega}) = \frac{d^2\Phi}{\cos\theta \cdot dA \cdot d\overrightarrow{\omega}} \tag{2}$$

where x is the position and $\overrightarrow{\omega}$ is the direction.

Radiance is the most important quantity in rendering simulation since it closely represents the color. Also radiance can be considered as the number of photons arriving per time at a small area from a given direction. Radiant energy can be computed by integrating the radiance field over all directions Ω and area A. The geometric setup for the radiance definition is illustrated in figure 2.2.

$$\Phi = \int_A \int_\Omega L(x, \overrightarrow{\omega})(\overrightarrow{\omega} \cdot \overrightarrow{n}) d\overrightarrow{\omega} dx \tag{3}$$

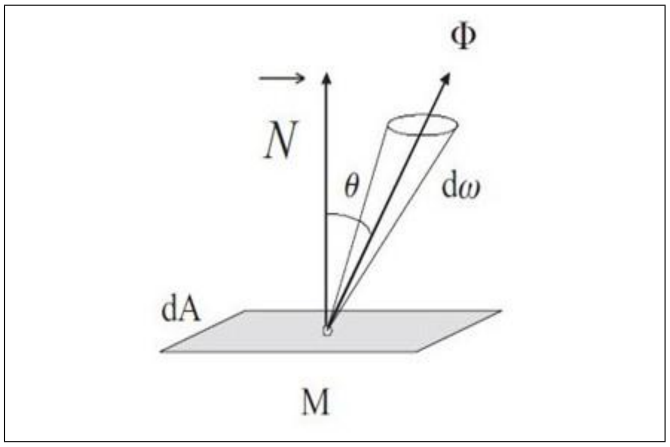


Figure 2.2: Radiance, L, is defined as the radiant flux per unit solid angle, $\overrightarrow{\omega}$, per unit projected area, dA

2.2 Fundamentals of Global Illumination

With the basic knowledge of radiometry, we will introduced a theoretical framework as the basis of global illumination. There are several components are included in this framework, light source, reflectance and visibility. We will look at these components firstly and then move on to the rendering equation describing the interaction between light and an surface with no participating media.

2.2.1 Light Source

As the light in the form of photons is emitted from light sources initially, the lighting becomes an essential input of the model. There are several types of light sources widely used in Computer Graphics such as point, directional and area lights. The radiance of lighting emitted from light sources is denoted as L_e.

We can measure the intensity of light source in *wattage*. Take a point light source for example, the power this light can emit is denoted by Φ, the emitted light distributes uniformly in all directions, and the irradiance, E, can be computed at a surface as:

$$E(x) = \frac{\Phi \cos\theta}{4\pi r^2} \tag{4}$$

where r is the distance from x to the light source and θ is the angle between the surface normal and the direction to the light source. From the equation we can intuitively tell a surface facing the source will receive more photons per area than a surface that is oriented differently.

2.2.2 Reflectance

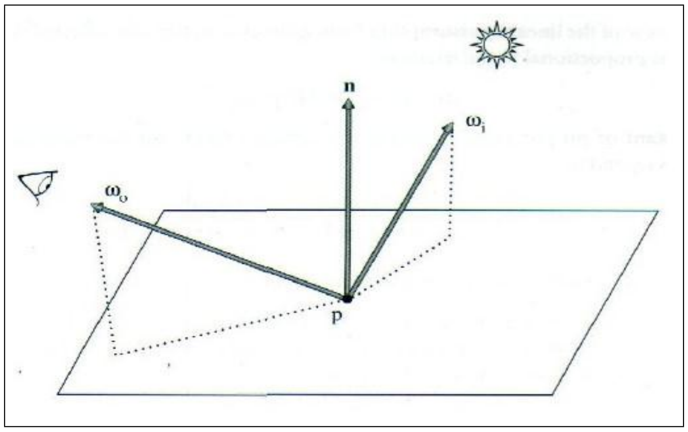


Figure 2.3: The geometric setup of BRDF.

The *Bidirectional Reflectance Distribution Function*, BRDF, is the mathematic tool describing the reflection of light encounters an surface. To define the BRDF, the geometric configuration is shown in figure 2.3. ω_i is the incident lighting direction, ω_o is the direction in which the reflected light leaving from the surface, and n is the normal vector at the location p on the surface. Given the incident radiance $L_i(p, \omega_i)$, we can find the outgoing radiance to the viewer, $L_o(p, \omega_o)$.

The BRDF, f_r, defines the relationship between differential reflected radiance and differential irradiance:

$$f_r = \frac{dL_o(p, \omega_o)}{dE(p, \omega_i)} = \frac{dL_o(p, \omega_o)}{L_i(p, \omega_i)(\omega_i \cdot n)d\omega i} \tag{5}$$

There are two important properties of BRDF used in rendering. The first one is the Helmholtz's law of reciprocity, that is given any pair of directions (ω_i, ω_o), we have:

$$f_r(p, \omega_i, \omega_o) = f_r(p, \omega_o, \omega_i) \tag{6}$$

Another important physical property of BRDF is energy conservation, stating that the total reflected energy is less than or equal to the incident energy. For all direction ω_o,

$$\int_\Omega f_r(p, \omega_i, \omega_o) L_i(p, \omega_i)(\omega_i \cdot n) d\omega_i \leq 1, \forall \omega_i \tag{7}$$

Note that there are special cases exists in the BRDF model, one is the Lambertian BRDF in which the outgoing direction is independent from the incident direction. Another extreme case is the perfect mirror reflection BRDF which is a Dirac delta function making the incident direction ω_i mirrored at the surface normal at the p on the surface. BRDF including these two special cases is often broadly classified as directional diffuse, and glossy and specular.

Given the definition of BRDF, we can introduce the basic rendering equation, also known as *local illumination model* by integrating the equation 5 over the sphere of incident directions around location p, the left side of the equation will be the outgoing radiance in direction ω_o.

$$L_o(p, \omega_o) = \int_\Omega f_r(p, \omega_i, \omega_o) L_i(p, \omega_i)(\omega_i \cdot n) d\omega_i \tag{8}$$

where Ω is the hemisphere of incoming directions at p.

2.2.3 Visibility

Another important component is the visibility computation. It is often modeled as a binary function denoted by $V(x, x')$ as shown in the following equation, where x and x' are two distinct points.

$$V(x, x') = \begin{cases} 1 & \text{if } x \text{ and } x' \text{ are mutually visible} \\ 0 & \text{otherwise} \end{cases} \tag{9}$$

The most important usage of the visibility test function is to determine whether a surface point is directly "visible" to a light source for correct lighting simulation. In a finite element algorithm like radiosity, the visibility between two surface locations is a essential factor in the rendering equation to compute the radiance leaving x in the direction towards x'. Further discussion on the rendering equation and radiosity will be presented in the following sections.

Ray casting is the most widely used operator to determine the closest surface in a direction by shooting a ray into the scene trying to find the closest intersection point with the surface. We will discuss ray casting and ray tracing technique further in section 2.3.2.

Another more complicated model of visibility is the non-binary function occurs between a surface point and an area light source, this model is used to simulate the soft shadows. [5] is a full survey of existing shadow methods dedicated to real-time rendering of soft shadows.

2.2.4 Light Transport Equation

The local rendering equation used to describe the direct lighting effect is too simple for simulating real-world lighting effect, so indirect lighting has to be introduced to this model as well. We introduce the Light Transport Equation (LTE) in this section to form the mathematical basis for all global illumination algorithms. The LTE is also known as the Rendering Equation (RE) which is introduced in the first time in [10]. We use the LTE term here to be distinguished from the local rendering equation, also it is more suitable with context of global illumination. The LTE states that the outgoing radiance L_o at location p on the surface in the direction ω_o is the sum of emitted radiance L_e and reflected radiance L_r. The reflected radiance can be computed using the local rendering equation 8 so we have final LTE as shown in equation 10.

$$L_o(p, \omega_o) = L_e(p, \omega_o) + \int_{\Omega} f_r(p, \omega_i, \omega_o) L_i(p, \omega_i)(\omega_i \cdot n) d\omega_i \tag{10}$$

2.3 Global Illumination Techniques

In this section, we will review the several the most popular approaches to compute the global illuminations including radiosity, Monte Carlo ray tracing and photon mapping, and the previous work related to these approaches.

2.3.1 Radiosity

Radiosity, also known as finite element approach, is a classic solution to solving the LTE. It was introduced by Goral et al. in [3] and has became an active field that was drawn quite a bit of research interest. The underlying idea is to tessellate the surfaces into finite small sub-surfaces called patches as geometric primitives and solve the LTE with them. Solving the LTE using radiosity requires several input. First of all the radiosity value for diffuse surfaces or the BRDF for the non-diffuse surfaces need to be stored for all the patches in the scene. Then it requires a known linear system of equations called form factors which denote the amount of light transported between two patches. See figure 2.4 for a figure showing the geometric aspect of the form factor. Before presenting the definition of form factor, a geometry term G is introduced with the following equation:

$$G(x, x') = \frac{(\omega' \cdot n')(\omega \cdot n)}{\|x' - x\|^2} \tag{11}$$

where x' is another surface location. We introduce the geometry term since for finite element algorithm the rendering equation is normally expressed as an integral over surface locations rather than ray directions used in ray tracing algorithm.

Thus the form factor F_{ij} representing the fraction of power leaving patch i that arrives at patch j is computed as

$$F_{ij} = \frac{1}{A_i} \int_{A_i} \int_{A_j} \frac{V(x, x')G(x, x')}{\pi} dA_j dA_i \tag{12}$$

where $V(x, x')$ is the visibility test operator introduced in the equation 9.

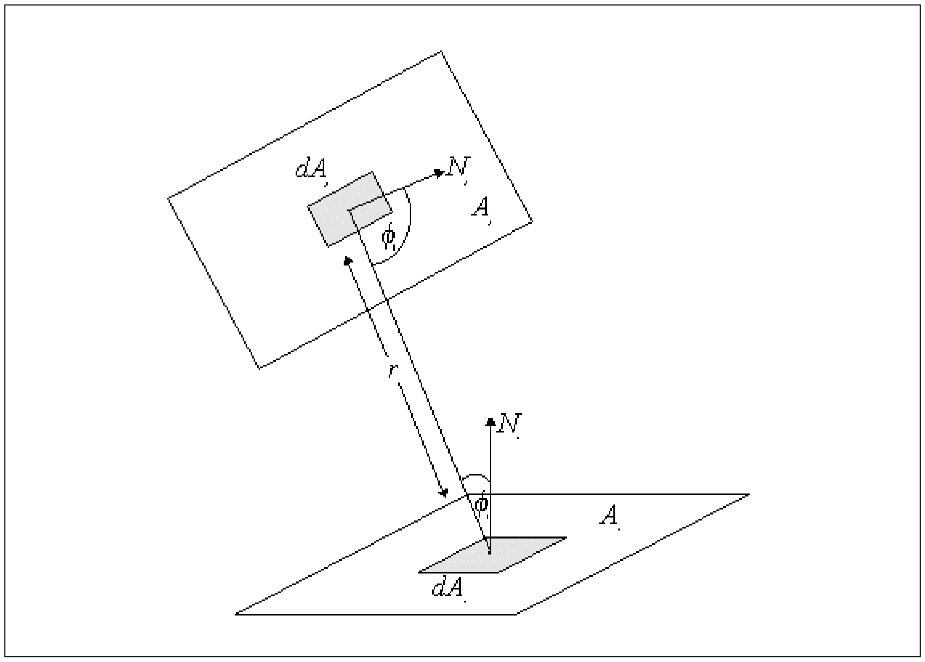


Figure 2.4: Geometry factor for two patches

Computing the form factors is the most time consuming part because of the initial time complexity of $O(n^2)$ of linking n patches, however when this pre-computation is done the entire scene can be rendered very efficiently. Therefore a lot researchers have focused on minimizing this performance hit. Hanrahan et al. [4] introduced a technique that constructs a hierarchical representation of the form factor matrix to reduce the computation. Holzschuch et al. [8] followed a hierarchical structure and introduced a progressive refinement strategy resulting the form factor only being computed when needed to evaluate the energy transfers from a given surface.

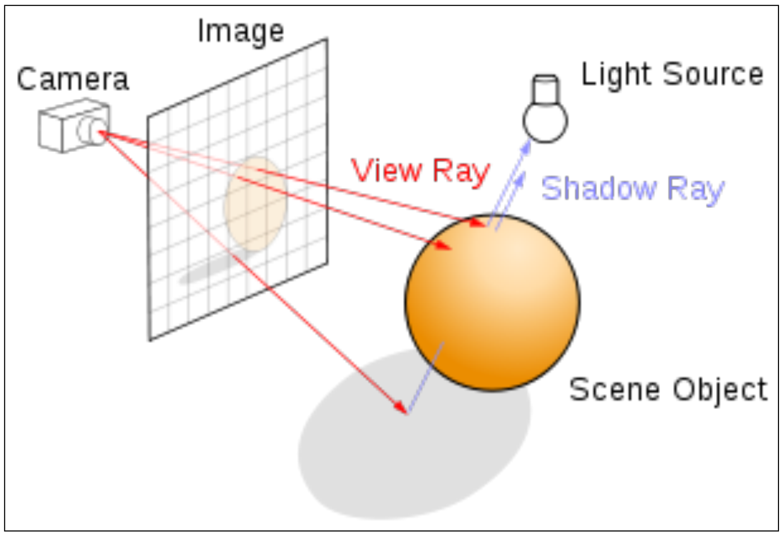


Figure 2.5: A basic configuration of ray tracing rendering system.

2.3.2 Ray Tracing

The classic ray tracing technique became popular in Computer Graphics since 1980 with an introduction of the recursive ray-tracing algorithm by Whitted [29]. It is an elegant and simple algorithm for easily rendering shadows and specular surfaces.

The ray tracing algorithm can be broke down into two stages: intersection query and shading. In the first stage, as shown in figure 2.5, for each pixel on our viewing plane, one or more rays (if the multi-sampling is enabled, this allows for better image quality) are shot into the scene. Those rays directly from the observer are *primary rays*. Then we try to find intersection point along each of the rays with the closest object to the observer. In the shading stage, at each intersection point the direct illumination is computed based on the BRDF determined by the surface material. The computed radiance is converted to the color of the corresponding pixel. The visibility of the light sources can be evaluated by casting rays from surface locations to the light sources trying to find any intersections with other objects other than lights, these rays are also known as shadow rays.

If the surface material is specular then a specular ray is traced in the reflected or transmitted direction. The indirect illumination can be computed by spawning and tracing reflected or refracted rays (called *secondary rays*) from the intersection points and repeating the ray tracing recursively.

Ray tracing is not a full global illumination algorithm since it cannot handle the computation of the indirect illumination on diffuse surfaces; it can only compute the illumination for perfect specular material by tracing a ray in the refracted or mirror direction. To simulate the phenomena such as soft shadows, it is necessary to employ Monte Carlo sampling techniques [10]. Figure 2.6 shows two images rendered by our rendering program – the scene on the right side contains an area light to produce soft shadows casted by objects rather than hard shadow produced with a point light in the left scene.

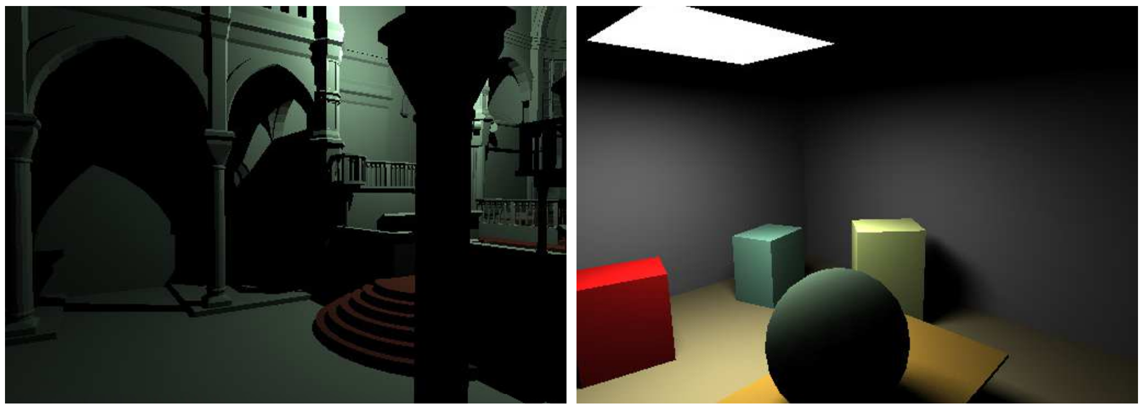


Figure 2.6: The scenes rendered by our test program using Monte Carlo ray tracing.

The idea of the Monte Carlo technique is to generate a large amount of samples of the rays and evaluate the LTE for every sample using the classic ray tracing technique and average all the results with the Monte Carlo integration to converge to the final solution.

Monte Carlo integration is a method of using random sampling to estimate the values of integrals. In order to estimate the value of integral $\int f(x)dx$ one needs only to be able to evaluate the integrand at arbitrary points in the domain.

The Monte Carlo estimator approximates the value of an arbitrary integral. Suppose $\int_a^b f(x)dx$ is an one-dimensional integral we want to evaluate, given a supply of uniform random variables $X_i \in [a, b]$, we can define the Monte Carlo estimator:

$$F_N = \frac{b-a}{N} \sum_{0<i<N} f(X_i) \tag{13}$$

and the expected value of F_N, $E[F_N]$, is in fact equal to the integral. This fact just takes a few steps to be demonstrated (see pp. 541-542 in [19]).

Monte Carlo ray tracing has several advantage over classic ray tracing:

- All global illumination effects can be simulated.
- Low memory consumption.
- Result is correct exception for variance (visible as noise).

The main disadvantage of Monte Carlo method is that it requires a huge amount of samples to minimize the errors. In order to reduce the errors in half, it requires evaluation of four times as many samples. Fortunately there are many techniques we can use to reduce the numbers of samples to compute while maintaining an acceptable image quality. A commonly employed method is importance sampling. Instead of generating samples rays blindly, we only send rays where the LTE's integrand has high values. This is easy for direct lighting but difficult for indirect lighting as the it is impossible to know where the largest illumination contribution comes from. Combining the other components like the BRDF f_r and incident lighting L_i into one importance sampling approach is more challenging for global illumination. Solutions to this issue includes multiple-importance sampling technique and bi-directional path tracing [11].

Accelerating ray tracing by exploiting the hardware have been an active research field as well. Optimizations for modern multi-cores CPUs for ray tracing renderer are presented [27].

2.4 Photon Mapping

2.4.1 Concept

Photon mapping was developed by Jensen [7] as an efficient alternative to the Monte Carlo ray tracing techniques especially for simulating the focused light effects, such as caustics. Photon mapping is a two-pass algorithm, photon tracing and radiance estimation.

Photon Tracing Photon tracing is the process of shooting photons from the light source(s) and tracing them into the scene similar to standard ray tracing. A global data structure is constructed to store the photons called the photon map. When a photon hits a diffuse surface, its position, incident direction and power will be stored in the photon map. Jensen suggests that using balanced kd-tree [1] to organize the photons data, since the kd-tree is beneficial to the next pass, radiance estimate. Whether the photon is absorbed or reflected is determined by the surface's BRDF. Multi-Sampling techniques such as Russian-Roulette [17] can be used to terminate this process earlier without damage the image quality. Photons that hit a specular surface will not be stored because the probability of have a incoming photons from specular direction is zero. Instead, these surfaces are rendered using standard ray tracing.

Radiance Estimate Given the photon map, we can perform a density estimate on certain surface points to calculate reflected radiance. The direct illumination and specular surfaces can be rendered using Monte Carlo ray tracing. As shown in figure 2.7, we collect n photons samples within a sphere make an estimate of the reflected radiance at any surface location x, as shown in equation 14.

$$L_r(x, \omega_o) \approx \frac{1}{\pi r^2} \sum_{0<p<N} f_r(x, \omega_{p,o}, \omega_i) \Delta\Phi_p(x, \omega_{p,o}) \tag{14}$$

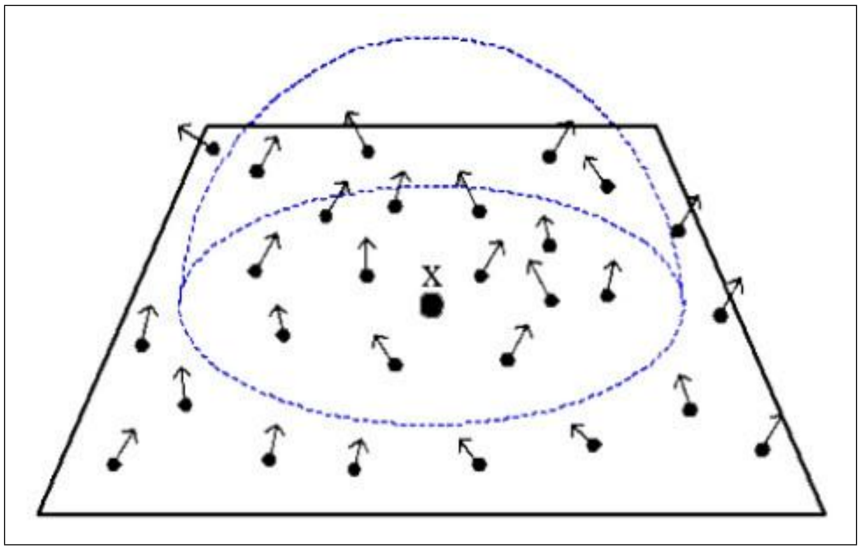


Figure 2.7: The geometrical setup of photons density estimation.

2.4.2 GPU-based Implementations

There have already been a couple of successful attempts made to implement global illumination techniques on GPUs. Purcell et al. [23] presented the first ray tracer entirely running on GPU using a uniform grid for acceleration. [9] and [21] presented the GPU implementations that achieved better performance than a CPU-based ray tracer. With the introduction of general purpose computing on GPU especially with CUDA technology, Luebke and Parker [14] presented a technology combining ray tracing and traditional rasterization methods to achieve real-time performance on dynamic scenes. Since the hardware-accelerated rendering pipeline is employed, their implementation could handle the scenes with moving light sources and dynamic scene objects by only tracing the rays directly cast from the camera (primary rays) and shadow rays, the full global illumination is still too slow for interactive application.

Photon mapping has been mapped for multi-core CPUs in [2] and for the older

GPU architecture using programmable vertex/pixel shaders in [24]. In the following sections, we will take a look at the techniques that can be used for performing photon mapping on current GPU hardware with CUDA programming model.

2.5 Photon Mapping On GPU using CUDA

In order to move the photon mapping technique onto graphics hardware, we need to do the creation of traversal of kd-tree on the GPU. The first attempt to achieve this was presented in [24]. However their work was limited by the GPU programming model and hardware architecture.

More general work on parallel kd-tree construction was presented in [20] and [25]. Both approaches are designed for multi-core CPUs thus are not suitable for GPU. The first problem is that kd-tree construction can easily become bandwidth limited on large input data sets due to its random memory access pattern. Therefore the construction switches from breadth first search (BFS) to depth first search (DFS) manner manner at deeper nodes. This means that these approaches keep the number of concurrently running threads pretty low. GPU hardware, however, is good at having a massive number of threads (at least $10^3 - 10^4$ running for optimal performance [18]. Another aspect is finding the split position for a node. Both papers use the Surface Area Heuristic (SAH) [15] to evaluate the costs for a splitting candidate. Even though the SAH improves kd-tree significantly [26], it is very time consuming.

Finally, parallelizing the photon search or tree traversing is relatively easy, as the tree is accessed read-only. However, for performance reasons it is important that the tree is well balanced and stored efficiently. Storing the tree efficiently means to keep scattered memory accesses as low as possible, by placing child nodes close to their parents. The traversal algorithm itself is not a good candidate for parallelization. Instead, performing multiple traversals simultaneously is a much better way, in order to obtain good performance.

Zhou et al. presented a new approach to kd-tree construction and traversal on GPUs using CUDA [30]. They also provide information on adapting the technique for photon mapping which we will focus on.

2.5.1 KD-Tree Construction

A kd-tree, standing for k-dimensional tree, is a space-partitioning data structure used for organizing geometry in a k-dimensional space by encoding the spatial information of geometric objects into a special case of binary tree to provide optimal search performance. The kd-tree used in Computer Graphics is usually 3-dimensional tree since the spatial information is modeled in 3D space. Construction of kd-tree involves recursively splitting the space with a chosen axis-aligned splitting plane. This process is usually done in a depth-first fashion with call the split routine recursively to create new nodes. However in the GPU implementation this depth-first approach is not ideal. Firstly the recursion is not supported currently by the GPU programming model, this issue can be avoided by transforming the recursion into iteration, but it is still not optimal since it does not take the advantage of the massive parallelism of the graphics processor. Therefore Zhou et al. build the kd-tree completely in breadth-first manner to avoid these issues.

During the initialization stage, CUDA's global memory is allocated for the tree construction and the root node is created. For the photon mapping implementation we also have to create three sorted order lists for each dimension for all point coordinates. The points can be grouped into chunks and we need to compute the tight bounding box of the current list of nodes, each node in this list contains certain numbers of chunks that are stored in a list as well. The first step to achieve this is to compute the tight bounding box of each chunk, this is a straight-forward process that only requires us to iterate over all the points in a chunk and find the minimum and maximum coordinates. The second step is to combine the bounding boxes of the chunks that belong to one same node to determine the bounding box of the node. This can be done by segmented reduction – the chunks that belong to the same node become a segment and we combine the bounding boxes on these segments respectively rather than combining over the entire list of chunks. Additionally we maintain three associated point ID lists (one for each coordinate axis).

The first stage of construction is called large node stage. In this stage splits nodes using a combination of spatial median splitting and "cutting off empty space" [6]. Since in this stage the number of nodes is small, it is better to perform the computation parallelized over all points rather than over nodes. First we need to find the splitting plane (the plane that splits the longest axis in the middle), after repeatedly applying empty space splitting before. Then each photon is classified as being either left (1) or right (0) of the splitting plane. Finally, we perform the scan operation [16]. After the split we check if the amount of photons in our child nodes is below the threshold $T = 32$. If the number is smaller, the node is added to the small node list. Otherwise it is added to the active list and scheduled for the next iteration of the large node stage. The large node stage finishes as soon as there are no more nodes in the active list.

The next phase is the small node stage. It begins with a preprocessing step for all nodes in the small node list. In this step we collect all splitting plane candidates and calculate the resulting split sets, which define photon distribution after a split. It should be noted that splitting planes are restricted to initial photon positions in this stage.

After the preprocessing has finished, we process all small nodes in parallel and split them, until each node contains one photon. Since we need to build a kd-tree for points (photons), rather than triangles, we are now using the Voxel Volume Heuristic (VVH) [28] for split cost evaluation, instead of the SAH [6].

After we found the best split candidate (the one with the lowest VVH cost) we can split the small node into two sub nodes. To do so we need the current node's photon set, which is a bit mask representation of the photons inside the node. In order to complete our split, we simply perform a logical AND between the current photon set and the pre-calculated result split sets of the root small node. As illustrated in figure 2.8, node A is split into two sub nodes(B and C) and the symbols represent the photons in node A.

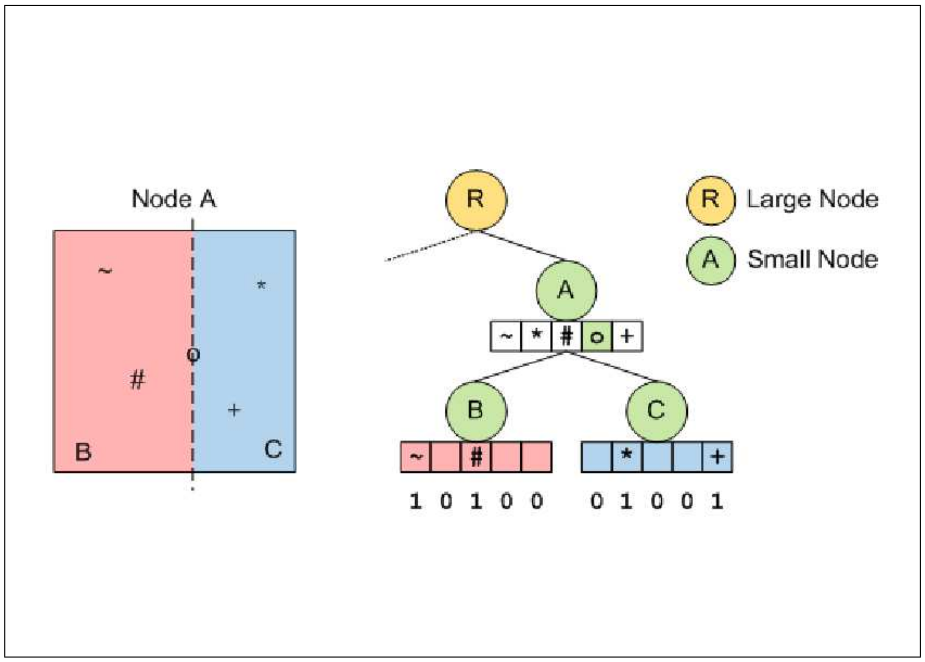


Figure 2.8: Small node splitting stage following the large nodes splitting stage in the kd-tree construction process.

Besides easing node splitting, the binary photon representation also helps us calculating the number of photons in a node, which we need for VVH computation and to stop node splitting. All we have to do is to count the bits in the current photon set, using the parallel bit counting routine.

After splitting is done, the new nodes are added to the active node list, in order to be processed during the next iteration step. Of course it can and will happen that some nodes will not create any new child nodes. Therefore, we have to add another step to compact our active node list and remove empty space, as is illustrated in figure 2.9 [12]. If there are no more nodes left in the active list, we can finish the small node stage and proceed to the final construction stage.

The final stage is the kd-tree output stage, the tree is reorganized to change its

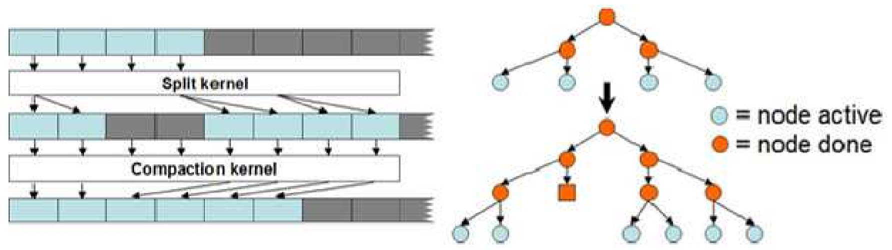


Figure 2.9: Following the execution of the kernel that splits the kd-tree nodes, a compaction process will be applied on the result nodes in the active node list for the next iteration over the list to build another level of kd-tree.

layout to a preorder traversal of nodes to improve memory access performance. Firstly we perform a bottom-top traversal to determine the size of the array for preorder tree structure. Then we use another top-down traversal to calculate each node's address and generate tree from sizes. The final output includes the node's bounding box, split plane and the references to its children and the photon's position and power.

2.5.2 Photon Search

To estimate the radiance at a surface point, the K-nearest photons around the sample point need to be located and filtered. In [7], Jensen presented a photon searching technique using the priority queue with kd-tree. Unfortunately it is not possible to implement a priority queue with CUDA efficiently since the memory access is incoherent and almost all arithmetic is independent, thus it makes difficult for hardware to hide latency. Therefore Zhou et al. propose an iterative K-Nearest Neighbor (KNN) range search algorithm based on [22], which is performed using a standard stack-based depth-first search kd-tree traversal algorithm. This algorithm can be efficient thanks to the fast CUDA's local memory for the stack. Instead of denoting the number of dimensions in kd-tree, K used here represents the target number of photons that are most nearest to surface point.

The algorithm starts from an initial conservative search radius r_0 and tries to find the query radius r_K through a couple of iterations. For each iteration, a histogram of photon numbers over different radius ranges is created and the final radius is reduced from it. The final radius r_K is then used for range search which returns all photons within that radius.

2.5.3 Advantages and Limitations

The GPU approach from Zhou et al. has a couple of advantages. One is that the KNN search is fast with the worst time complexity of $O(k \cdot n^{1-\frac{1}{k}})$ [13], where $k = 3$ for a three dimensional kd-tree. Another advantage is the compact data arrangement of the tree giving a space complexity of $O(n)$.

However this approach also suffers from a couple of limitations. First of all, the dynamic lists are extensively used for storing active nodes, small nodes, final tree nodes and so on and only static arrays are used for optimal performance, therefore additional work has to be done to grow the lists by reallocating the memory and data copying which is expensive operation. To avoid the memory management overhead Zhou et al. double list sizes when the array need to grow. This leads to increased memory consumption during construction and a big performance hit as we need rebuild the kd-tree for photons every frame. Another limitation is the complexity of implementation, there are too much temporary data to maintain during construction and the requirements of many other data primitive algorithms such as scan, split and sort also increase the complexity of this approach.

Chapter 3

A New Approach

In this chapter, we will present a novel approach for rendering a scene with dynamic light source efficiently. This method is based on the standard GPU-based photon mapping rendering system, using an augmented kd-tree data structure and localized updating algorithm for rendering.

In the first 2 sections of this chapter, we will present an overview on our approach and a comprehensive description of the data structure that we use and related algorithms. Then we will look at some of the GPU implementation details.

3.1 Motivation

As described in chapter 2, we build a global kd-tree as the photon map for radiance estimation. In comparison, the standard approach suffers from a main limitation that the global kd-tree for photons must be rebuilt from scratch for every frame.

This process is time consuming due to the complexity of kd-tree building algorithm on GPUs. Our approach avoids this costly overhead.

3.2 Overview

Instead of building a global kd-tree for photons, we re-use the same kd-tree for the geometric objects (triangles) for efficient radiance estimation using KNN search having one kd-tree organizes the spatial information of both geometric objects and photons distribution in the scene. Similar to the way how triangle data is associated with a kd-tree leaf node, we create a new structure to store both photons and kd-tree leaf nodes data to enable us quickly performing the photons search with kd-tree traversal algorithm and update the photon data in this structure without rebuilding a separate kd-tree for photons.

For each frame, the photons are shot from the light source and stored in an array, then we build the kd-tree for the geometry objects if it is required, for the scenes that don't include animated objects we will skip this process. Given the photons data in the scene and the built kd-tree for geometry, we build our data structure, compress the memory to bind it to the texture memory for a better memory accessing performance. In rendering phase, instead of using another kd-tree for a global photon map for radiance estimation, we use the kd-tree and our photon queue to perform the KNN search. Updating of the kd-tree for scene is not required since it is static, therefore we only need to perform a traversal, when the leaf nodes are reached, we

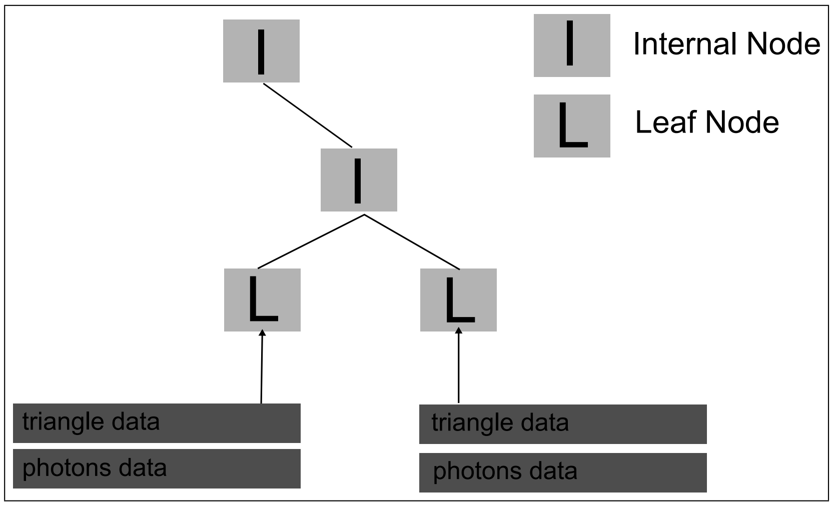


Figure 3.1: The photons data will be stored with a connection to the kd-tree for the scene.

will have the access to the photons associated with this node, utilizing two indices on the array to indicate the range of current frame's photons, we can directly perform a photons search.

The update process of photons queue is straightforward. It depends on the photon data from the previous frames. Here we keep track of all the photons data of a range of frames with a pair of indices, the indices can be updated and maintained efficiently.

Further details of the data structure and the update process will be presented in the following section.

3.3 Data Structure and Algorithm

3.3.1 Data Structure

As shown in figure 3.1, in addition to the triangle data, the photons data in the scene is also logically associated with the leaf nodes of the kd-tree. As the kd-tree for the

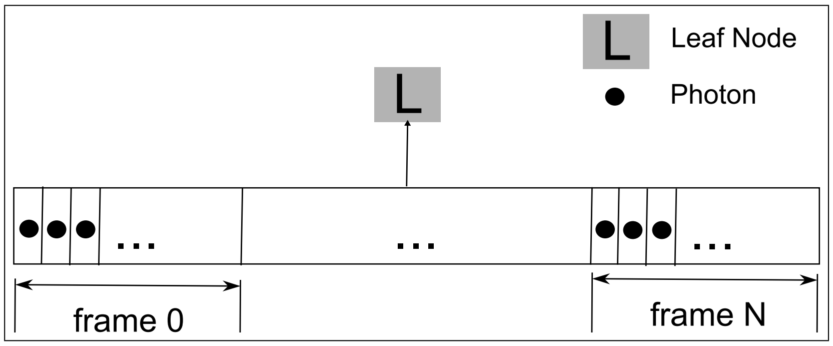


Figure 3.2: More detailed view on how photons data is organized for a single kd-tree leaf node. There will multiple frames of photons stored in a continuous block of memory.

scene actually encodes the spatial relationship among the geometry, and the photons will be stored when they hit the objects with diffuse objects, we attach the photons fall into the spatial region occupied by the bounding box of a kd-tree leaf node.

In figure 3.2 we have a more detailed view on how the photons data organized. The photons shot into the scene in one frame of rendering are stored followed by the photons of next frame. When implementing this organization on GPU, all actual photons data (positions, incident directions and power) is stored in a separate array, the indices of the photons associated with kd-tree leaf nodes are stored instead of the actual photon data.

In figure 3.2 we show a more detailed view on how the photons data organized. The photons shot to the scene in one frame of rendering are stored followed by the photons of next frame. When implementing this organization on GPU, all actual photons data (positions, incident directions and power) is stored in a separate array, the indices of the photons associated with kd-tree leaf nodes are stored instead of the actual photon data.

3.3.2 Construction

Building the logical connection between photons and leaf nodes is simpler than build a global kd-tree for the photon map. Firstly, we found out the the leaf nodes from the built kd-tree in parallel. This can be done checking a flag set when building the kd-tree. Given the indices of leaf nodes, we can retrieve the bounding box from the small node list generated in kd-tree construction phase(see 2.5.1). Then we check which leaf node certain group of photons should fall into in parallel by performing fast point-box intersection testing. For each leaf node, the number of photons can be calculated with a parallel reduction operation. Further details on physical memory arrangement and implementation of the data structure will be presented in section 3.4.

3.3.3 Update

When a frame of image is rendered, new photons will be shot into the scene and be stored in the same global array with previous frames and for each leaf node, we need to keep track of the range of the photons that are active for rendering the next frame. For each leaf node, we maintain two pointers (indices), start and end index (in the sense of a circular array), indicating the range of the photons used for rendering. We move the end pointer forward when there are new photons arriving, move the start pointer forward when there are some old photons we need to discard. We use a pre-defined threshold from count to determine how many frames of photons data we want to make active, we keep accumulating the photons every frame until we reach that threshold value. When the threshold frame count is reached, we move forward the start pointer to avoid there more photons than the threshold. Since the number of photons per frame is known, the stride of moving start pointer can be calculated. However, in GPU implementation, the step in bytes used to increment the pointer is usually larger than the exact size of photons data we want to discard, since the memory will be padded when it is allocated for better memory accessing performance. Further details on the implementation and coding will be presented in section 3.4.

3.3.4 Rendering

Rendering with our data structure does not differ much from the standard photon mapping rendering algorithm. In rendering phase, we cast the rays from the camera into the scene in parallel and perform KNN search using the kd-tree already built for the geometry, when we reach the leaf nodes we use our data structure to find the photons data for particular leaf node and gather the photons for the radiance estimation.

3.4 Implementation Details

3.4.1 Data Organization

KD-Tree Data The kd-tree data should be carefully organized when implemented with CUDA to improve traversal performance. In 2.5.1, we have already described the kd-tree building algorithm introduced in [30], but some of the details that are critical to the program's overall performance still need to be discussed here.

The final node list generated when the kd-tree is built is not sufficient for fast traversal algorithm, since it contains too much useless information. This is not friendly to cache pre-fetch. Every time when we are trying to access the memory of next kd-tree node's data, the memory next to the location we access to will be pre-fetched to the high speed on-chip cache for better performance. If the kd-tree nodes data is not compact, the instruction being executed will fail to read the data in the cache.

Therefore we compress and re-organize the traversal related data of the kd-tree nodes to reduce the memory access. The entire kd-tree is stored in a structure of several contiguous arrays. The structure is defined as following:

```
struct kdtree_data
{
  // Number of nodes
  uint  num_nodes;
  // Size of d_preorder_tree in bytes
  uint  size_tree;
  // Number of elements for each node
  uint* d_num_elems;
  // Addresses for each node in d_preorder_tree
  uint* d_node_addresses;
  // Traversal Information
```

```
    uint* d_preorder_tree;
    // Extent for each node.
    float4* d_node_extent;
};
```

The traversal data is compressed into an unsigned integer array, as shown in the following figure:

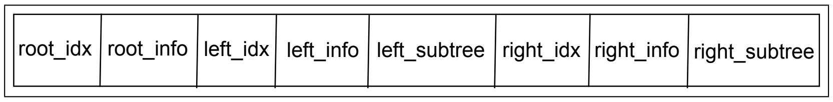


Figure 3.3: Compacted kd-tree data memory layout which is friendly to kd-tree traversal.

In the array, `root_info`, `left_info` and `right_info` are the parent information for the root, left and right child node (if left and right node are inner nodes). This is a compressed form of what is needed during kd-tree traversal. Parent information stores two unsigned integers, hence two elements in the d_preorder_tree array. The first unsigned integer contains several information of the inner node: the split axis takes 2 most significant bits and address of right node in the d_preorder_tree array, which is 0 if it is a leaf node, take the rest bits. The second unsigned integer stores the split position (a float value stored as unsigned integer). The address of left child node is not stored explicitly as it can be computed by skipping the parent info(2 unsigned integers) from current node's address. The 2 bits for split axis can represent x axis if it is 0, y axis if it is 1 and z axis if it is 2.

Also, in the above array, `root_idx`, `left_idx` and `right_idx` are the indices of the root, left and right nodes. The indices can be used to access the other arrays such as the node extents array. The type information of the nodes is also encoded in the indices, if it is a leaf node, the most significant bit (MSB) is set, else the MSB is not set, this improves leaf detection performance because it avoids reading child information. The above format applies only for inner nodes, for leafs, instead of parent information, element count and element indices are stored. The element indices are relative to the underlying triangle data. Hence we have the following format shown in figure 3.4:

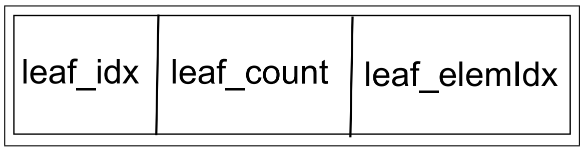


Figure 3.4: The compacted memory layout of kd-tree leaf nodes.

Photons Data

Connection Between KD-Tree and Photons Data The connection between kd-tree leaf nodes and photons data is stored similarly to the memory layout shown in figure 3.4. Instead of triangle data, the indices of photons traced into the scene of the current frame is stored as the elements.

3.4.2 Algorithm Description

Construction

Algorithm 1: Classify photons to kd-tree leafs.

```
input  : KD-Tree Data
input  : Photons Data
output: KD Leaf Nodes Photons List
leafNodesMarks ← new array;
leafNodesIndices ← new array;
identityIndices ← (0, 1, 2 ... n);
for All nodes of KD-Tree in parallel do
    leafNodesMarks ← mark leaf nodes;
end
leafNodesIndices ← new array;
tempArray ← Multiply identityIndices and leafNodesMarks;
leafNodesIndices ← Compact tempArray;
for All KD-Tree leaf nodes in parallel do
    for All photons of current frame in serial do
        Perform point-box intersection detection.
        Store indices of photons
    end
end
for All leaf nodes of KD-Tree in parallel do
    if frame counter < max frames then
        Store photons indices of current frame with current end pointer.
    end
    Update start and end pointer for current leaf node.
end
```

In algorithm 1, two major pieces of input data are required, the kd-tree data constructed for the scene and the photons data shot in the scene. The photons data is updated every frame by inserting new photons into a large chunk of buffer resides in GPU memory. Similar to the organization of underlying triangle data for kd-tree nodes, the memory for each frame of photons is aligned to a fixed size to enable coalesced memory access which may improve the performance considerably. This is one of the most important good practices for CUDA programming, because if the threads in a block are accessing consecutive memory locations, then all the accesses are combined into a single request by the hardware. However, organizing data in this way will bring extra complexity for implementation because we have to keep track of the locations of beginning of next valid chunk of data for data accessing and there will be holes contained in the whole data array. Therefore move the pointer from ith frame to $i+1$ frame requires increasing the pointer by maximum size of memory of ith frame which is aligned to a fixed size.

Before going into the description of the algorithm 1, we will give an overview on the fundamental parallel algorithm which are serving as the building blocks of parallel application, then we will see the application of these parallel algorithms with the description of our algorithm.

Parallel Reduction

Parallel reduction is a common data parallel algorithm that generates a single result value from many parallel threads. It can be illustrated with the figure 3.5:

Parallel Compact

Another common algorithm is parallel compact, the results are output per thread. It is often used to remove the null elements. It can be illustrated with the figure 3.6:

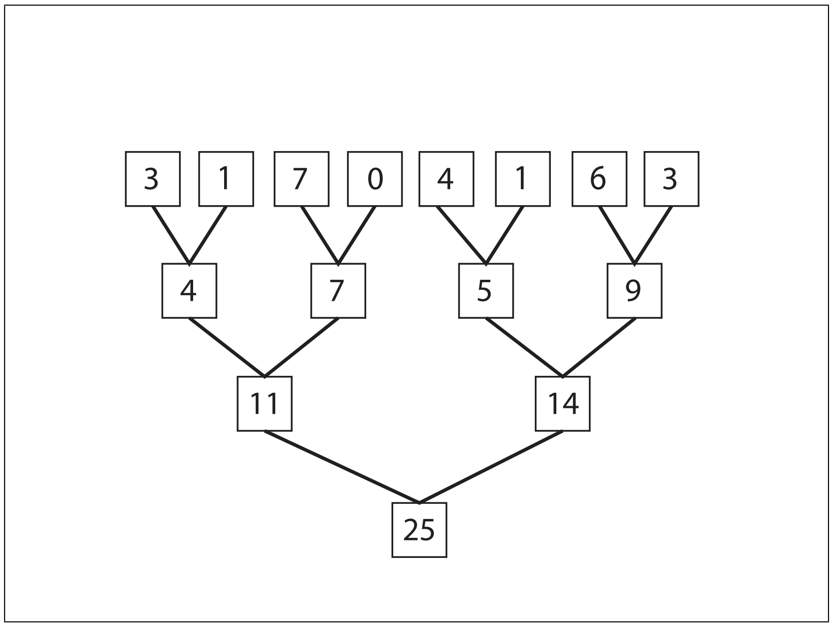


Figure 3.5: Parallel reduction using "+" operator.

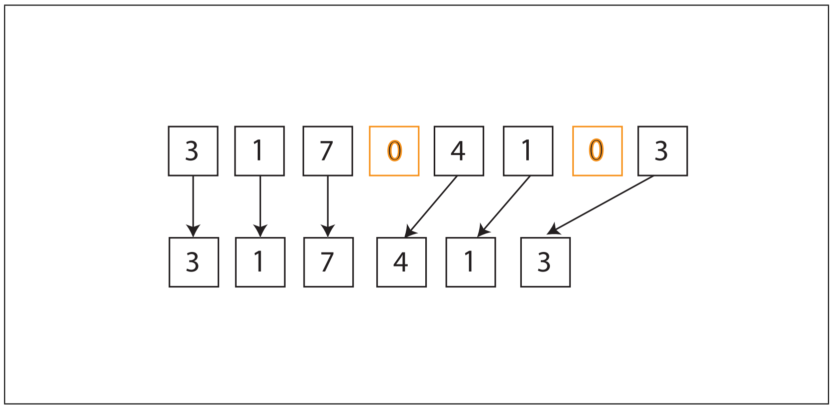


Figure 3.6: Parallel compact to remove null elements.

We firstly mark the leaf nodes from the input kd-tree data, given the data definition described in the section 3.4.1, we only need to look at all the nodes and test if the MSB of the node's index is set. The indices of nodes can be retrieved from the kd-tree data. We store the result of marks in an array with the same number of elements as number of kd-tree nodes, the result will be 1 for a leaf node and 0 otherwise. The number of leaf nodes can be calculated by performing a data parallel reduction on this array. In order to store the indices of leaf nodes into an array of our data structure, we use another temporary local array and fill it with identity values $(0, 1, 2...n - 1)$, where n equals to the total count of kd-tree nodes, then we use the marks of leaf nodes as a filter to select the leaf nodes by performing an array multiplication for each elements on these two arrays, lastly we need a data parallel algorithm, compact, to remove all the invalid values (0s) from the result of previous step and store the result. The entire process is illustrated is the figures 3.7 and 3.8.

Our goal is to connect photons with kd-tree leaf nodes so that we can easily find the photons according to the indices of leaf nodes. Therefore we need to classify the photons into an array with the same order of the indices of leaf nodes. For example, the ath leaf node associates with a range of photons $[i, ..., i + n]$. Firstly, we launch a kernel program on all leaf nodes performing the intersection detections between the bounding box and point on all of photons, if the photon is contained in a certain leaf node, then we store the indices of photons, figure 3.9 illustrates the process.

The data structure used to store photons for kd-tree leaf nodes works in the similar way to a circular buffer. We store number of frames of photons in a contiguous buffer.

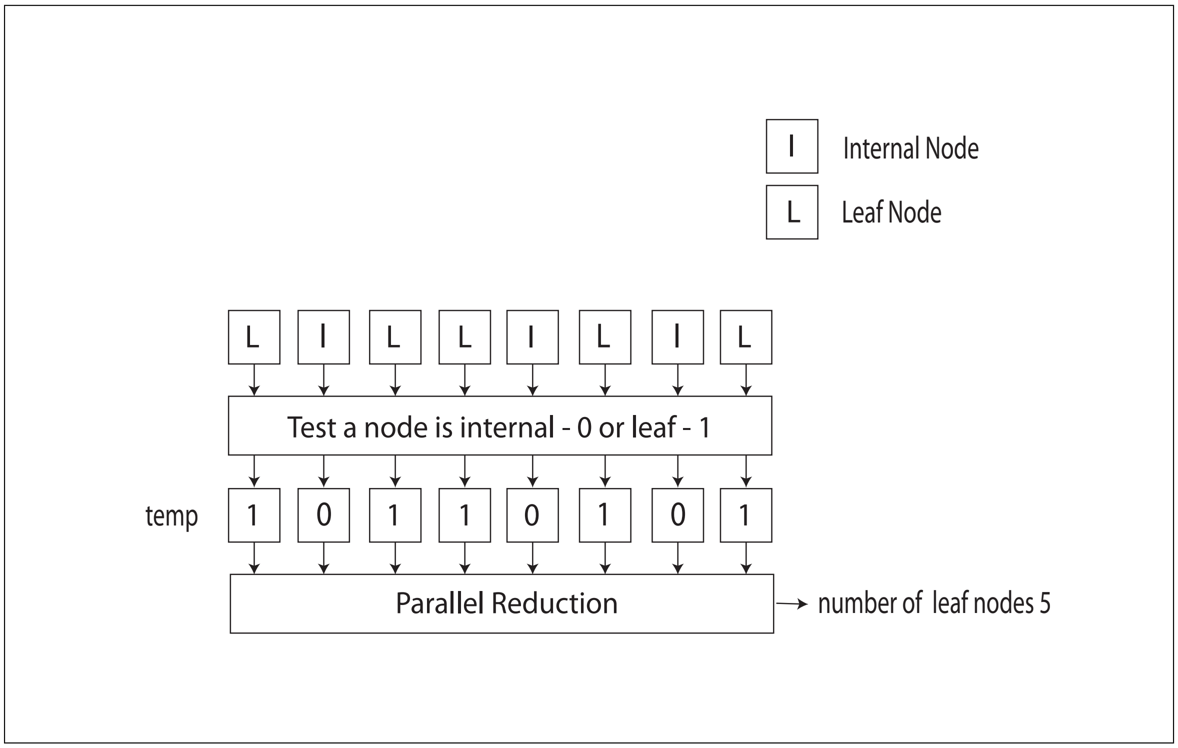


Figure 3.7: Find number of leaf nodes using parallel reduction.

The start and end pointer (indices) indicate a range of frames of photons that is valid for rendering this frame. The start pointer points to the first photons of first active frame and the end pointer points to the first photon of the last active frame. We have a frame counter indicating the number of frames we have in the range, therefore we can detect if the buffer is full by compare the frame counter and max frames capacity.

Render

Algorithm 2 shows the main framework of estimating the radiance at a query point in parallel. Each CUDA thread works on one query point, see figure **??**. The key part of this computation is performing KNN range search around the query point with the kd-tree and the photons queue as input. The KNN range search function sums up

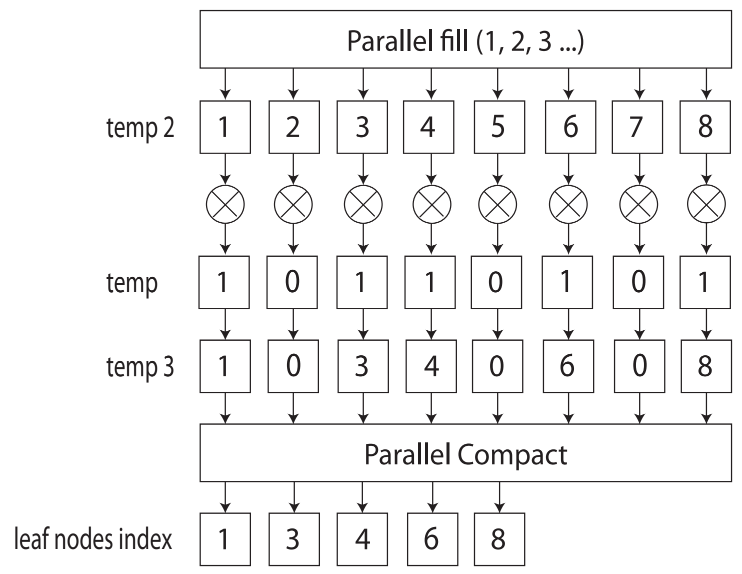


Figure 3.8: Find indices of leaf nodes in parallel.

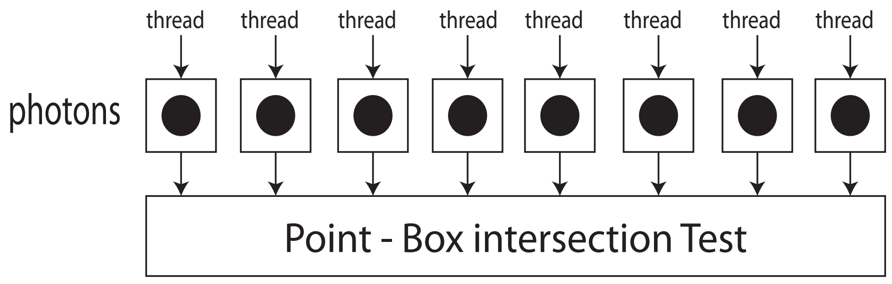


Figure 3.9: Photons-box intersection tests in parallel.

Algorithm 2: Radiance estimation with kd-tree and photons queue.

```
input  : Query Point
input  : Camera Rays
input  : Query Radius
input  : Diffuse Colors
output: Radiance
for All intersection points in parallel do
    Retrieve diffuse color at intersection point;
    Retrieve the normal vector at intersection point;
    Flip the normal vector if it is needed;
    irradiance ← rangeSearch(intersectPoint);
    radiance ← colorDiffuse $\cdot\pi^{-1}\cdot$ irradiance;
end
```

all the power (flux) of photons and output the irradiance by dividing the flux by the area of circle centered at the query with a predefined radiance.

In algorithm 3, we present the KNN range search using our data structure. The basic layout of the algorithm follows the standard while-while iterative traversal scheme. We first allocate an array as the stack for the kd-tree traversal. Note that this array resides on the local memory of the CUDA thread running this algorithm, therefore the access to local array is always coalesced. Then we visit the kd-tree nodes in pre-order style using the compressed kd-tree data. If the current node is an internal, then we determine which subtree we will go deeper by compare the position of query point and split plane with respect to the split axis. If the other child node is in the range of search radius, it is going to be pushed to the stack. When a leaf node is reached, we look up the photons queue in which the range of photons for searching is marked by the start and end pointer and iterate over the photons finding the ones in the searching radius and calculate the irradiance.

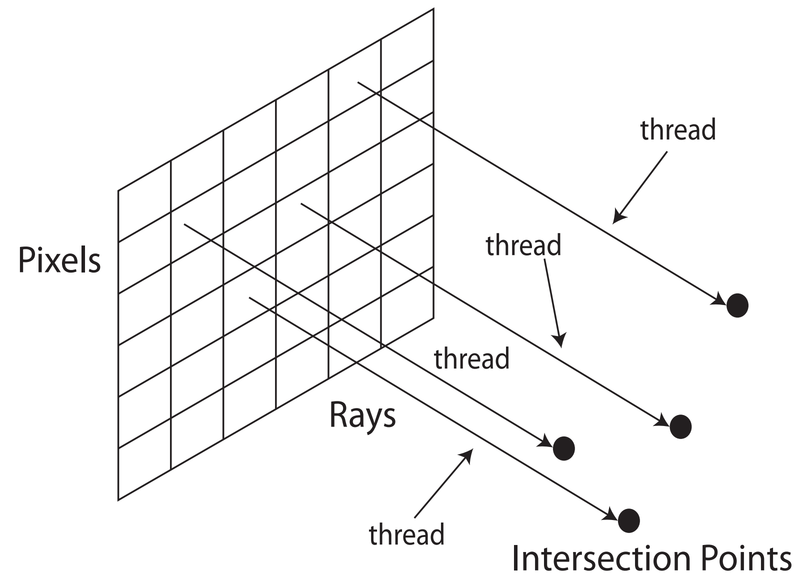


Figure 3.10: Each CUDA thread works on one query point.

Algorithm 3: Range search with kd-tree and photons queue.

```
input  : Query point position and radius
input  : KD-Tree Data
input  : Leaf nodes photons queue
output: Number of photons found
totalFlux ← 0;
todoStack ← new Array[KD_MAX_HEIGHT];
nodeAddr ← 0;
while nodeAddr >= 0 do
    idxNode ← kdTreeData.preorderTree[nodeAddr];
    isLeaf ← idxNode & 0x80000000;
    idxNode ← idxNode & 0x7fffffff;
    while NOT isLeaf do
        leftNodeAddr ← nodeAddr+1+2;
        parentInfo ← GetParentInfo(kdTreeData);
        splitAxis ← GetSplitAxis(parentInfo);
        splitPos ← GetSplitPos(parentInfo);
        rightNodeAddr ← GetRightNodeAddr(parentInfo);
        distSqr ← Sqr(queryPoint[splitAxis] − splitPos);
        nodeAddr ← leftNodeAddr;
        otherNodeAddr ← rightNodeAddr;
        if queryPoint[splitAxis] > splitPos then
            nodeAddr ← rightNodeAddr;
            otherNodeAddr ← leftNodeAddr;
        end
        if distSqr < queryDistSqr then
            todoStack[todoPos++] ← otherNodeAddr;
        end
        Read node index + leaf info (MSB) for new node.
    end
    /* Now we have a leaf node */
    photonIdx ← photonsQueue.start;
    numPhotons ← photonsQueue.numPhotons;
    while has more photons do
        photonPos ← GetPhotonPos(photonIdx);
        pointDistSqr ← ComputeDistantSqr(queryPoint, photonPos);
        if pointDistSqr < queryDistSqr then
            totalFlux ← Add the flux of photons;
        end
    end
    nodeAddr ← Pop next node from next stack;
end
return irradiance ← totalFlux  maxRadiusSqr * π;
```

Chapter 4

Simulation Result

In the first section, we will have an overview on our simulation program and our testing environment. Then in the following section, the simulation results and analysis will be presented as concept proof of our new approach.

4.1 Overview

Our simulation program is implemented with CUDA, the host part of the code is fully written in C++ and the code running by GPU is written in CUDA C which is an extension of standard C programming language. All of the code for GPU kernels is encapsulated in separated files (*.cu).

Several 3rd party frameworks are employed for a variety of features. Open Asset Import Library (Assimp) is an open source library to import various well-known 3D models formats. It is written in C++ and very easy to integrate with our code.

Another important framework we used is wxWidget. wxWidget is C++ based cross-platform light-weight Graphics User Interface (GUI) toolkits. We use it for the user interface (UI) and displaying the final render result.

4.1.1 System Specification

The testing system's specification is listed in the table 2:

CPU	Intel Core i7 2620M 2.7GHz
System Memory	8GB DDR3
GPU	nVidia Quadro 2000M
GPU Memory	2GB DDR3
CUDA Cores	192
CUDA Version	4.0
Driver Version	296.88

Table 2: System Specification

4.1.2 Application Specification

In our test program, the following key features are implemented:

- GPU-based(CUDA) Monte-Carlo ray tracing and kd-tree implementation.
- GPU-based(CUDA) photon mapping technique.
- GPU-based(CUDA) photon mapping with kd-tree leaf nodes photon queue.
- Use CUDA-OpenGL interop for the final render result display.

We use the CUDA Toolkit 4.2 and Microsoft Visual Studio 2012 as the base CUDA development environment, the build target is `release` and all default optimizations are turned on.

The approach that we make comparison with is standard GPU-based photon mapping. Two kd-trees are built for both ray-tracing acceleration structure and global photon mapping. The implementation of the kd-tree construction and photon searching is based on the algorithms in [30]. For each frame, all the photons of the light source are shot into the scene and the kd-tree for the photon map is re-constructed. In the following we will use the the term "standard approach" approach to refer to this implementation.

4.2 Data Structure Construction

Firstly, we will look at the performance of the data structure construction. Given a total number of photons shot into the scene, we compare the construction time among a range of frames. As shown in figure 4.1, the construction time of the standard approach (red curve), the red curve indicates the construction time with the standard approach , that is we rebuild the kd-tree for photon map every frame from scratch with all the photons shot from the light source in the scene. The green curve represents the construction time using our incremental update approach. For each frame, we trace fewer photons (500 photons instead of 5000 in this test case) and keep accumulating the photons for rendering the following frames. Also the complicated kd-tree construction process is avoided. Therefore we can see the data structure construction time is shorter than the standard approach.

4.3 Memory Consumption

The memory consumption is another important aspect we need to investigate. The memory capacity of the PC's video cards nowadays has been greatly extended, it is fairly easy to find an affordable video cards featured with 1024MB DDR5 on-board memory. However video memory is still a type of valuable resource. When analyzing the memory consumption of our technique compared to the standard approach, we are looking at the data from two different running phases of our test program, the pre-rendering phase and rendering phase.

As shown in figure 4.2, given the parameter of *max frames* as *10*, the memory consumption of our technique is considerably larger than the standard approach, in fact, it requires more than 10 times more memory. The reason for this behavior is that we maintain a large buffer that works like a circular buffer to keep track of all the photons data of the previous frames, and the capacity of the buffer is determined by maximum number of frames we check back. In addition to the extra photon data, we also need to store another buffer on GPU memory that stores the photons that belongs to a kd-tree leaf node using photon indices which point to the photon data, buffer. Though the index is just represented with an 32-bit unsigned integer value

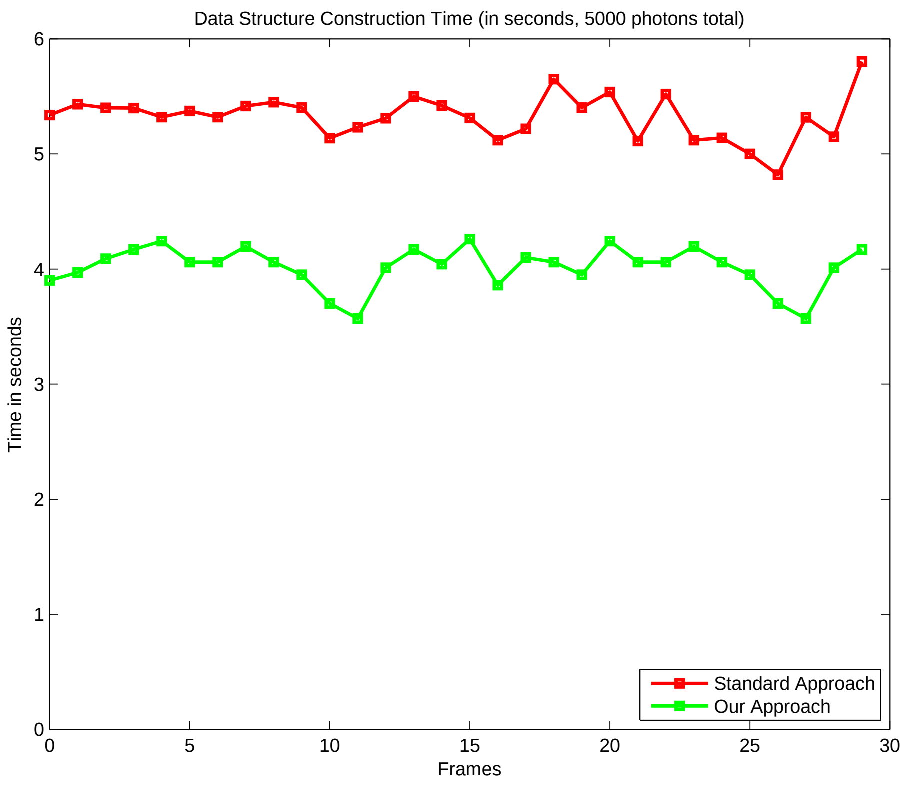


Figure 4.1: Data structure construction time measured in seconds.

which is much smaller than the detailed representation of a photon in the global buffer.

Although our new technique demands much more memory than the standard approach does during the rendering phase, the memory requirement of our method overall is much lower than the standard approach. This behavior is due to the large memory overhead introduced in kd-tree construction phase. Large amount of memory is allocated for the miscellaneous data structure such as the node list, the indices of triangles (triangles and points) and split list, and then will be released when construction is done. Conversely our new technique does not require much temporary data.

As shown in figure 4.3, building a kd-tree for the photons demands much more memory than building the photons queue, even though the memory allocated here will released later, the peak memory consumption could bottleneck the performance of the whole program or even prevent the program from running for some video cards with smaller memory capacity. Since memory allocation is an expensive routine and kd-tree construction requires lots of temporary memory allocation/release, it becomes another important factor that brings negative impact on performance of the data structure construction for the standard approach as we can see in the section 4.2.

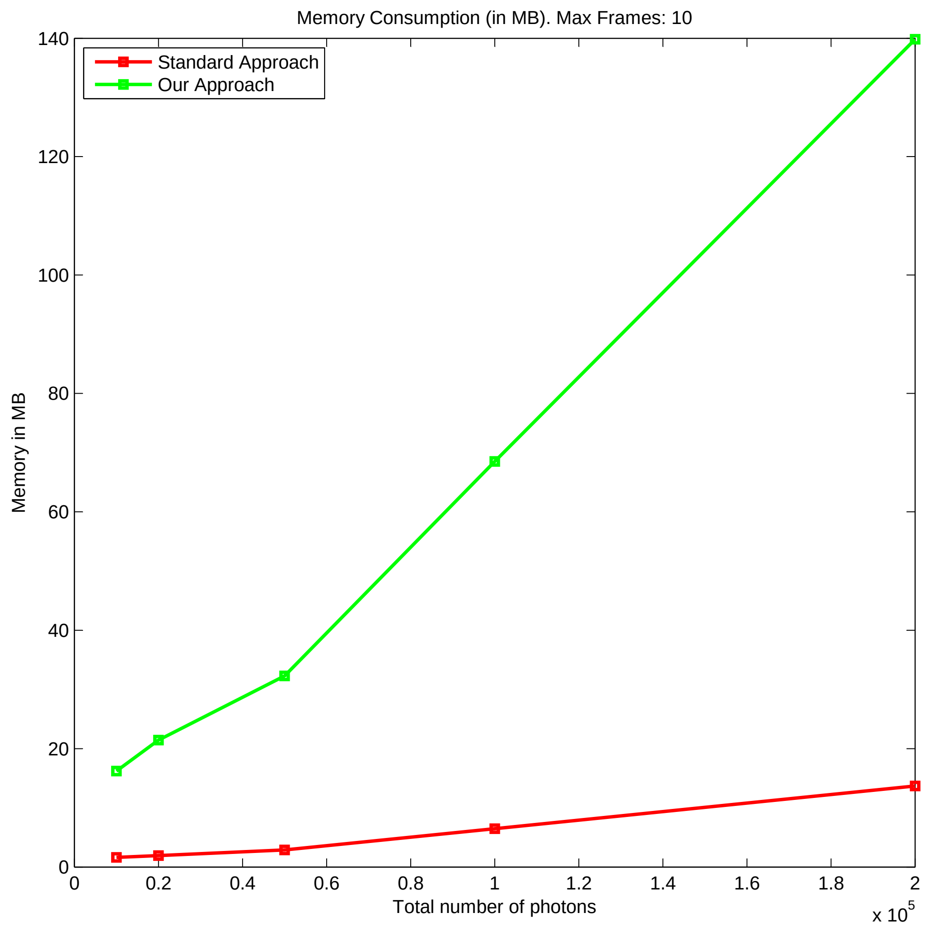


Figure 4.2: Memory consumption in rendering phase.

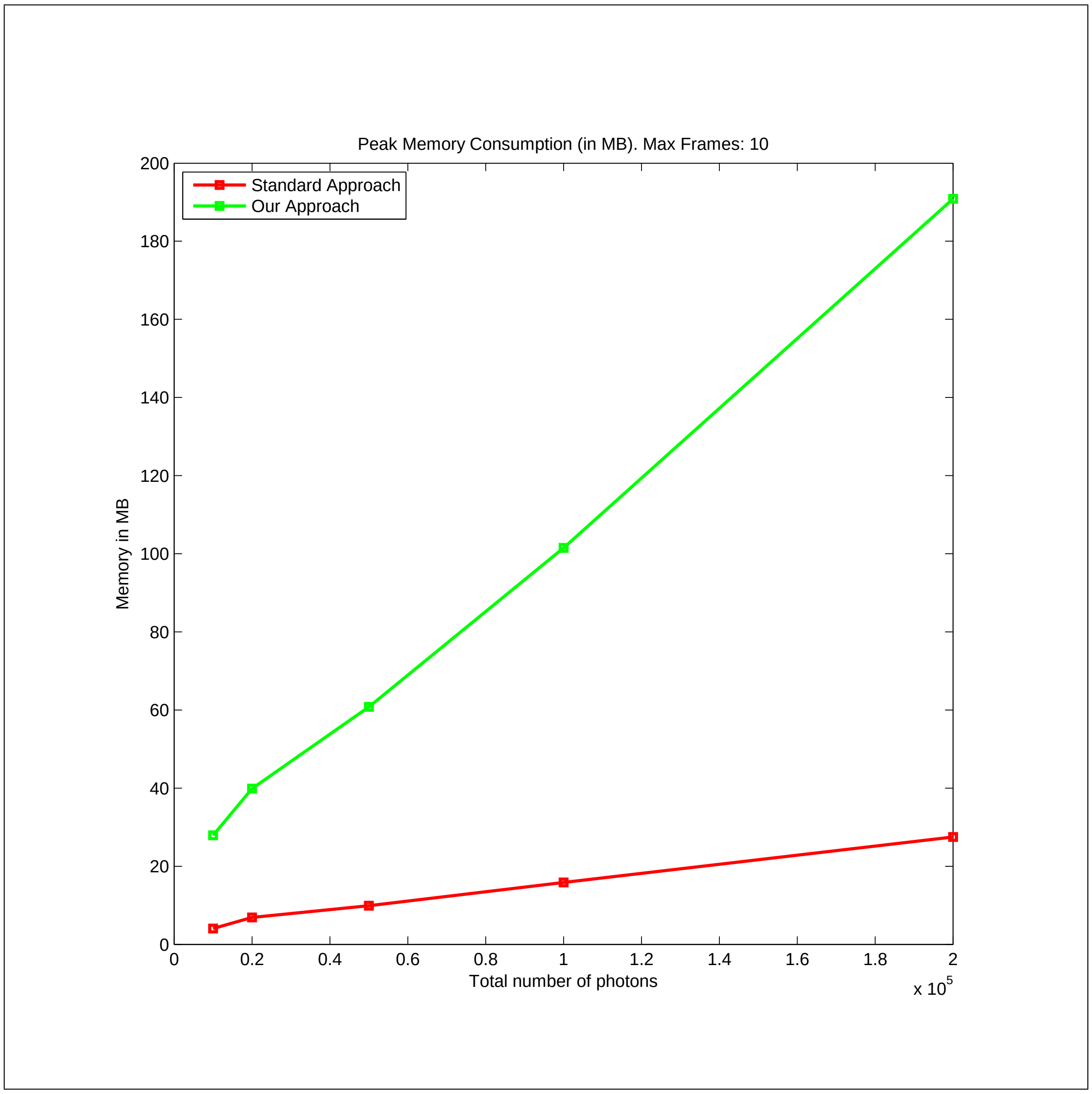


Figure 4.3: Memory consumption in construction phase.

4.4 Photons Search

In this section, we will show how the different techniques perform during the photon search. There are couple of parameters that have influence on the performance:

- The number of photons to search.
- The search radius(maximum allowed distance between the query point and a photon).

We will present how these parameters effect the photon search result and analyze the possible cause.

4.4.1 Number of Photons

As shown in the figure 4.4, the photon gathering of our technique is slightly faster than the standard photon mapping when number of photons is less than 150000, as the range of photons to search grows, it slows down and is eventually out-performed. As introduced earlier, our technique creates a buffer holding the photons that spatially belong to the leaf nodes of kd-tree for the scene objects, when performing the photon search, we traverse this built kd-tree with k-nearest neighbor algorithm to quickly reach the leaf nodes that is connecting with the photons that potentially fall into a pre-defined search range from the query point using the bounding box information of the kd-tree leaf nodes, then perform a linear search at this level. Compared to a kd-tree built for the hundreds of thousands of photons in the scene, the kd-tree for geometry is usually far more coarse, especially for a relative simple scene. Therefore the kd-tree for a simple scene will have a small depth and a leaf node could have an large extent and holds a large number of photons. Thus the advantages of the binary search is out-weighted by the linear search. This is a possible reason why we have the result shown in figure 4.4. In an extreme case, when there is no split in the scene, thus the kd-tree only has one node, the root node which contains everything, and the photon search will degenerate into a linear search.

4.4.2 Query Radius

Figure 4.5 shows that how the query radius effect the photon search performance. The standard photons search algorithm and our technique both have an almost steady increase in search time with an increasing query radius. For standard photon mapping method, since more kd-tree node will be enqueued to the stack waiting to be visited as the query radius increases, this brings larger kd-tree traversal overhead. When using photons queue for range search, the lost of performance becomes less than the standard photon mapping. As explained earlier, more photons are associated with one kd-tree leaf nodes, thus there are less nodes pushed to the local stack and the overhead of traversing the kd-tree is avoided.

4.5 Image Quality

In this section we take a look at the quality of our approach. As described in chapter 3, our new approach uses an incremental scheme to update the kd-tree and photons

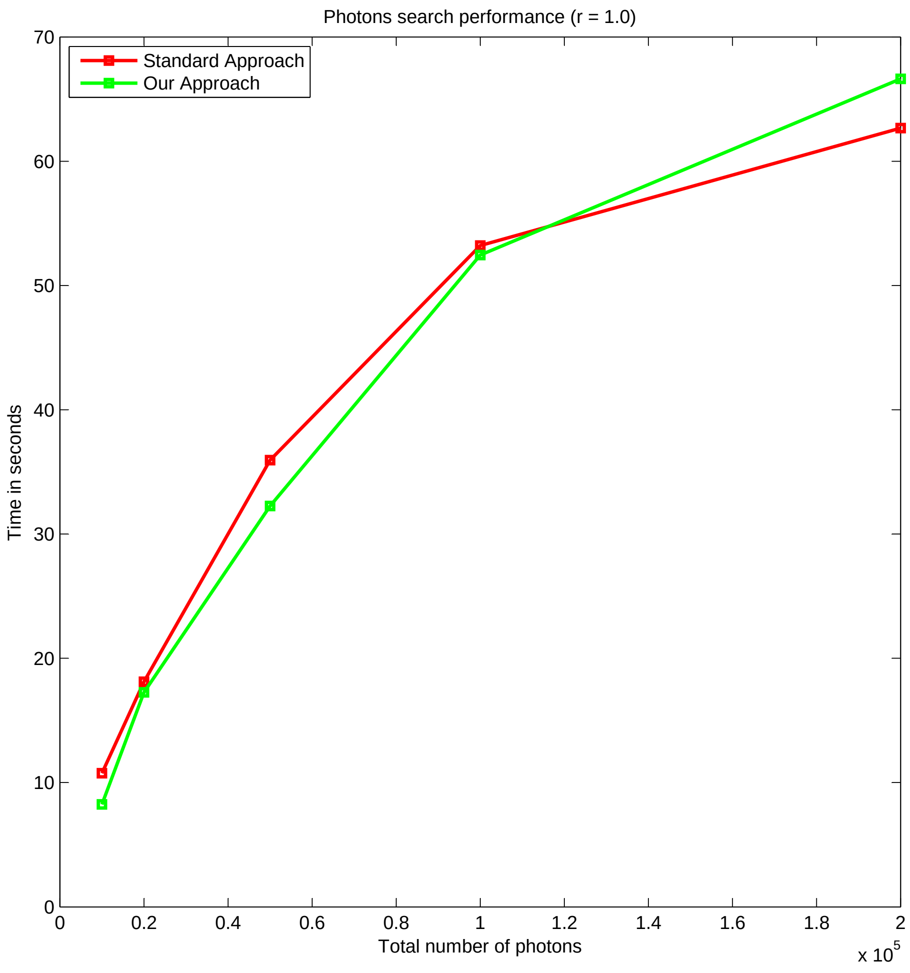


Figure 4.4: Photon search performance with different number of photons.

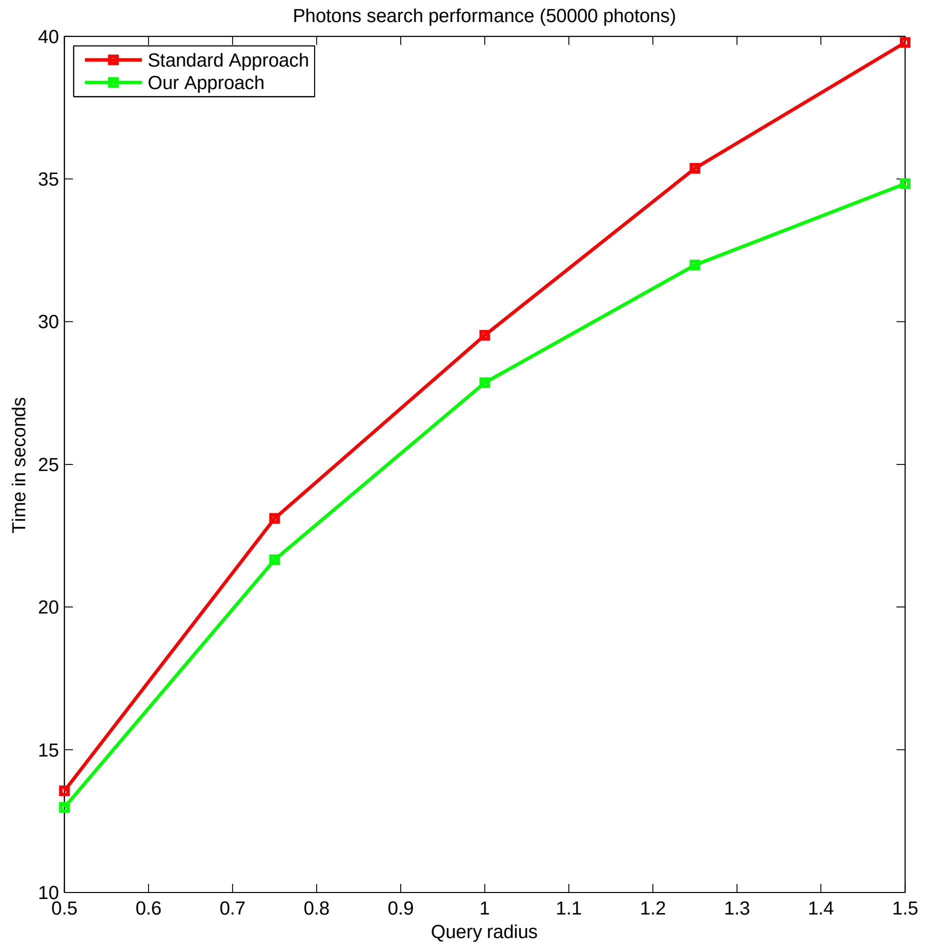


Figure 4.5: Photon search performance with different query radius.

queue for the rendering to be beneficial for the dynamic scene with moving light source. Therefore it makes more sense to look at the result images of a range of frames to see the image quality, the light source will also be moving in the frame sequence so that we can see the change of equilibrium distribution of radiance in the scene. To obtain a better knowledge on how the photons are distributed in the scene each frame, we will present the photons visualization result images as well along with the result image.

Several key parameters defined in our experiments for both the standard photon mapping method and our new approach are presented in the table 3.

In figure 4.6, we firstly present the result image rendered with default values of the parameters listed in table 3 using standard photon mapping algorithm and photons visualization image.

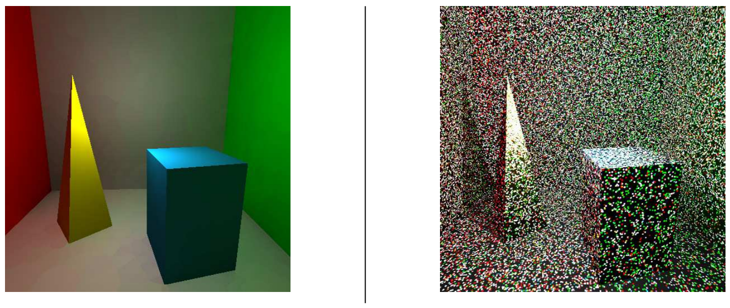


Figure 4.6: Result images rendered using standard photon mapping technique (left) and photons visualization image (right).

The following series of image groups rendered with our new approach is a frame sequence rendering a scene with moving point light source.

As shown in figure 4.7, figure 4.8 and figure 4.9, the quality of resulting images

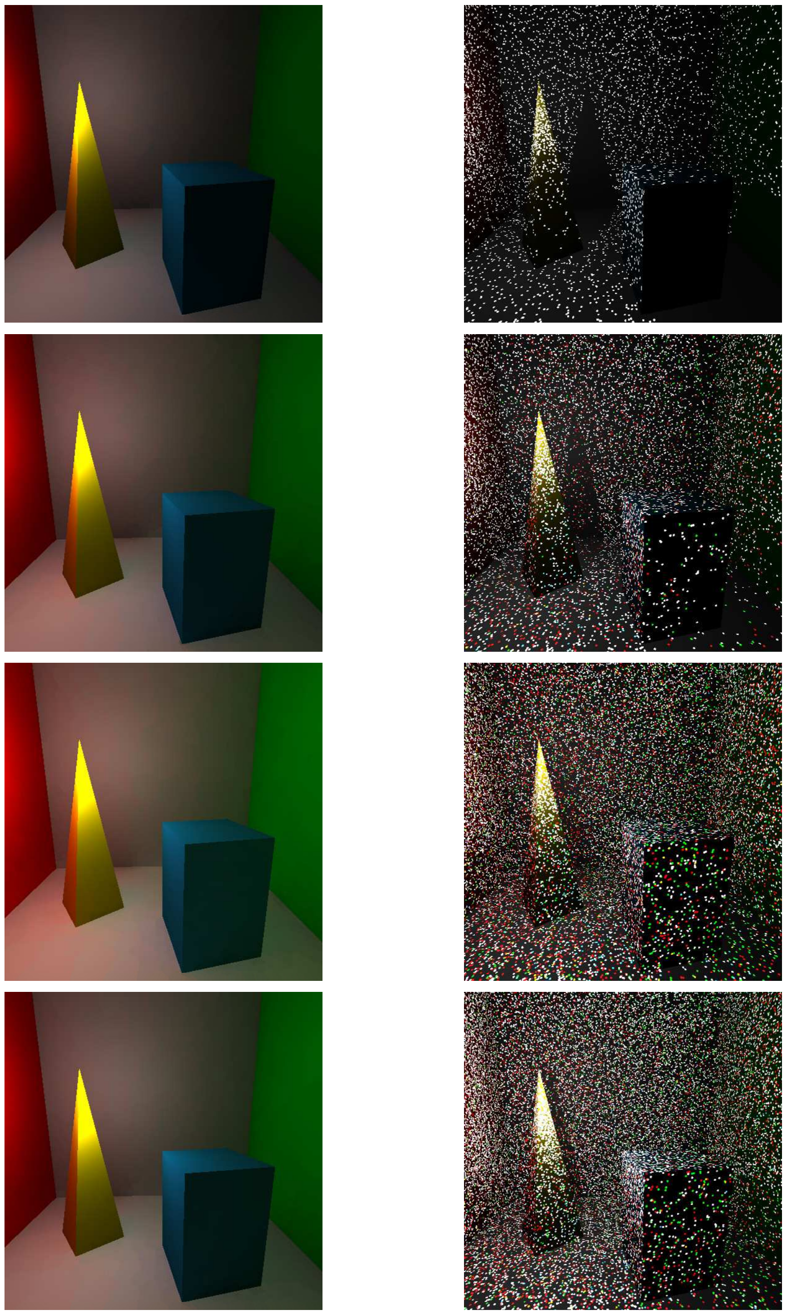
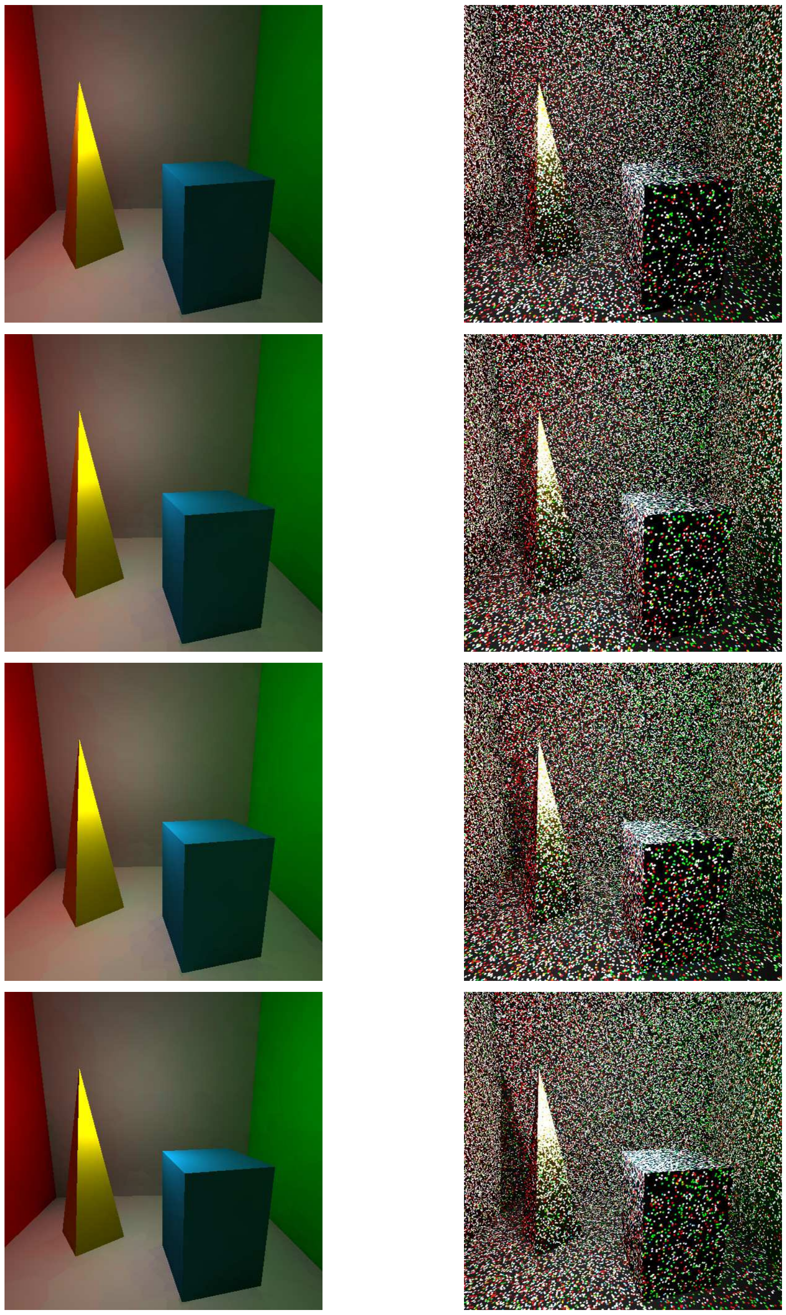


Figure 4.7: Rendered images and photons visualization from frame 0 to frame 3.



(a) (b)

Figure 4.8: Rendered images and photons visualization from frame 4 to frame 7.

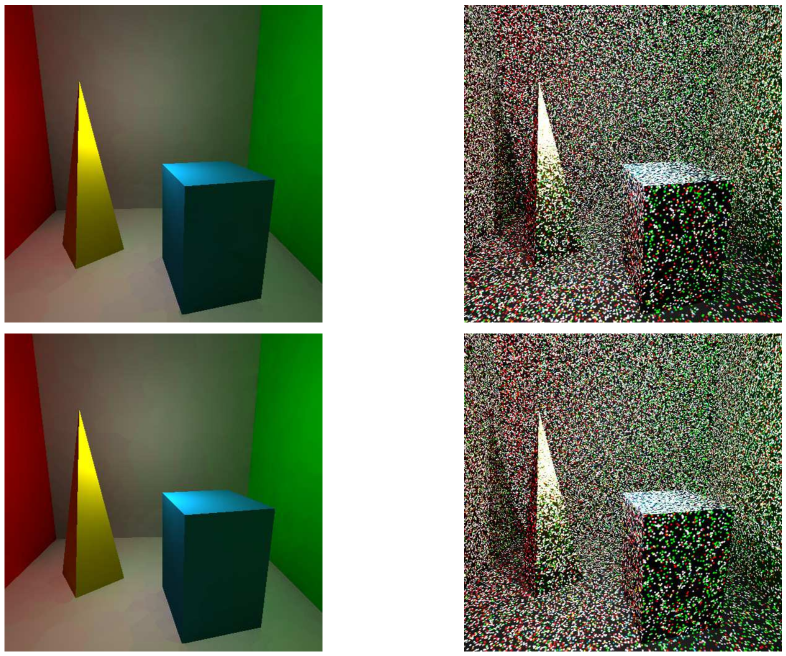

(a) (b)

Figure 4.9: Rendered images and photons visualization from frame 8 to frame 9.

become better and finally stable after a certain number of frames. It is not difficult to understand why we have this result according to our algorithm, as we trace fewer photons for each frame, the beginning few images apparently are less illuminated by the light source, in the following frames, we keep shooting more and more photons into the scene and accumulating the photons to the photons queue. Therefore we can visualize more and more photons and the scene becomes brighter. The number of photons in the scene will become stable when the maximum frames, since there are same amount of photons traced as they are supposed to be compares to the standard approach. Eventually we will achieve same image quality as that of the standard photon mapping.

Figure 4.10 shows that the images rendered by our approach produces the same quality as of the standard approach.

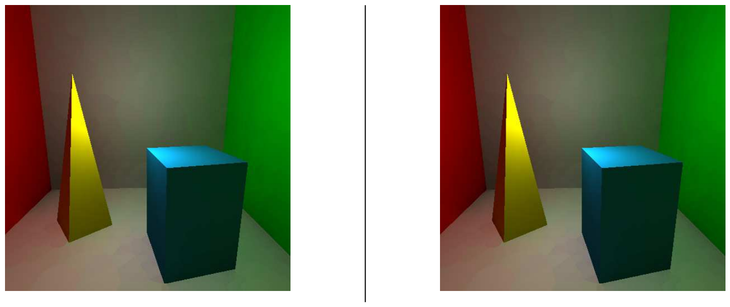


Figure 4.10: Rendered images and photons visualization using standard approach (left) and the our new method with full amount of photons (right).

In the following group of tests, we present how the image quality changes with the moving light source using our approach. We move the light source from the world coordinates (-6, 10, 6) to (6, 10, -6) using 4 frames and move it back to (-6, 10, 6), it takes 7 frames in total. While the light source moving, our new renderer updates the kd-tree to store more photons, it will have the maximum number of photons which is 200000 at 4th frame and continue updating photons queue by moving forward the start index to maintain the proper range of photons. We will make a side-by-side image comparison with the standard approach for each frame.

In figure 4.11, we can see that from frame 0 to frame 3, the scene on the left is less illuminated since there are less photons shot into the scene than there is supposed be emitted from the light source. We use these 3 frames to prepare the photons queue by update it incrementally. In the last frame, the image quality is similar to that of the standard approach since there are full amount of photons stored. Figure 4.12 shows that our new technique can achieve as good image quality with the dynamic light source as the standard approach after finished preparation of the photon queue and started using the start and end pointer for the radiance estimation. There are more frames shown in figure 4.13 to illustrate the images quality using our approach to incrementally update the photons queue for rendering.

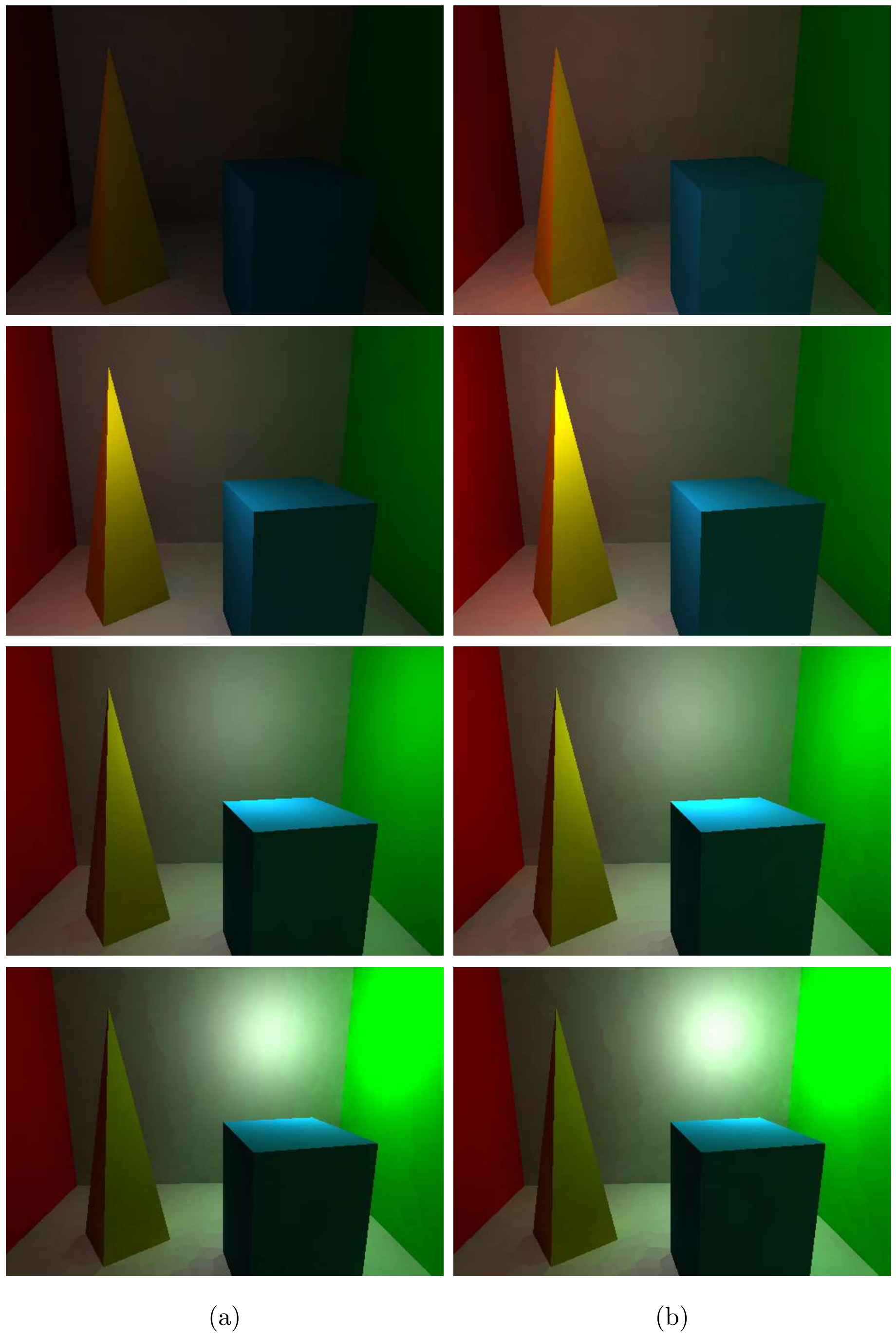
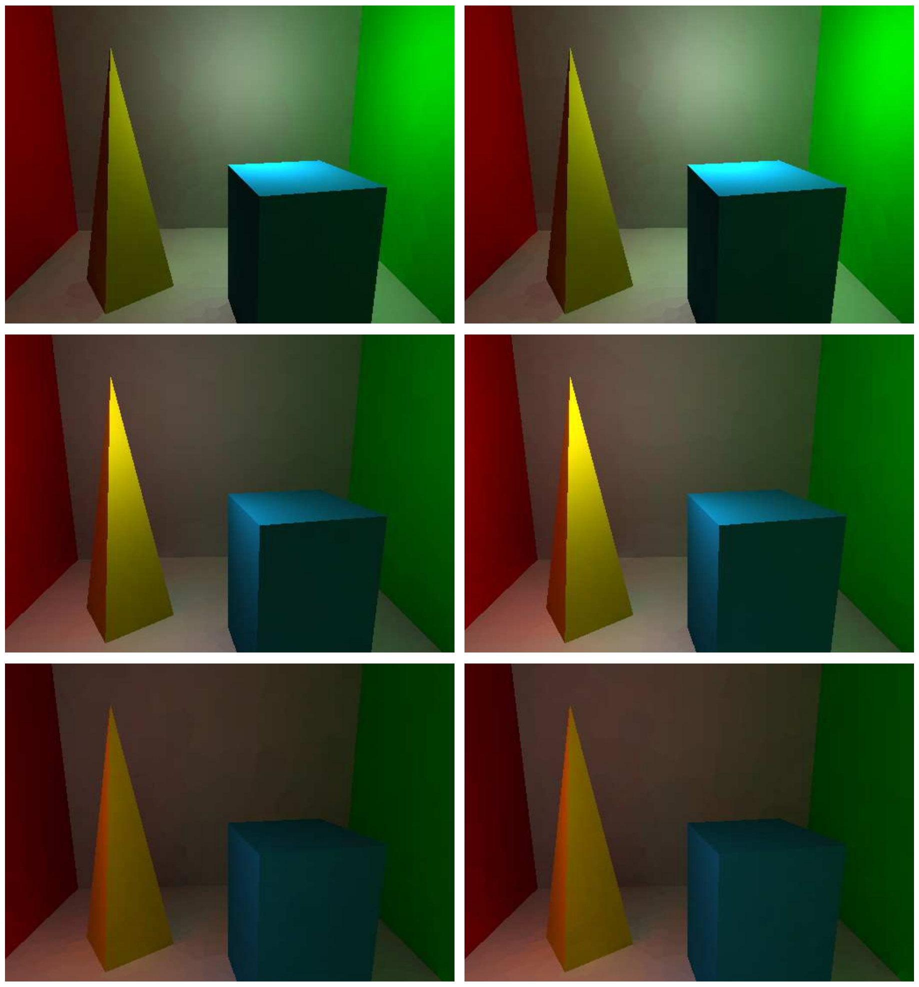
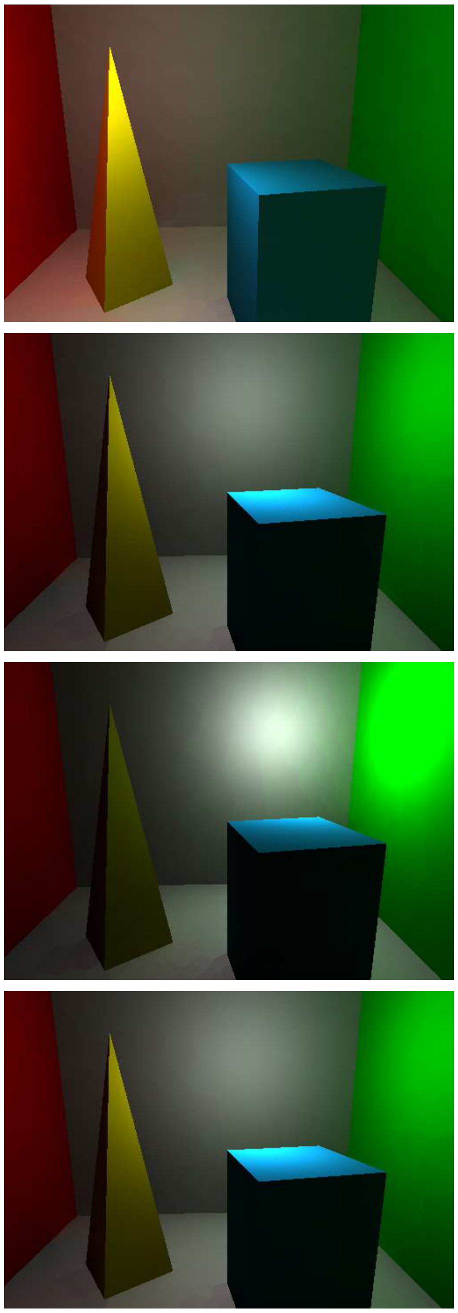


Figure 4.11: Comparison of image quality from frame 0 to 3 with moving light for the new approach (left) and standard method (right). Corresponding to each frame the positions of the light source are (-6, 10, 6), (-2, 10, 2), (2, 10, -2), (6, 10, -6). The number of photons maintained by our photons queue are 50000, 100000, 150000 and 200000 for each frame.

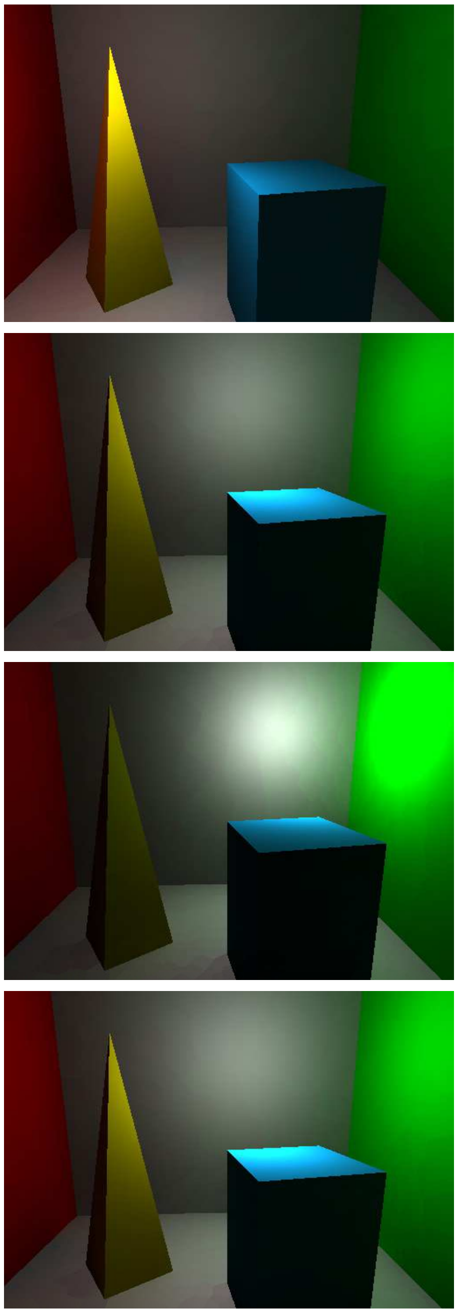


(a) (b)

Figure 4.12: Comparison of image quality from frame 4 to 6 with moving light for the new approach (left) and standard method (right). Corresponding to each frame the positions of the light source are (2, 10, -2), (-2, 10, 2), (-6, 10, 6).



(a)



(b)

Figure 4.13: Comparison of image quality from frame 7 to 10 with moving light for the new approach (left) and standard method (right). Corresponding to each frame the positions of the light source are (-2, 10, 2), (2, 10, -2), (6, 10, -6), (2, 10, -2).

Table 3: Experiments Parameters

Parameter	**Default Value**	**Description**
Max. Photons Bounces	3	The maximum photons reflection will be traced.
Total Number of Photons	200000	The total number of photons emitted from the light source.
Light Source Type	Point Light	The type of light source, point light and area light are supported. However in this experiments only point light is used.
Light Source Position	(0.0, 10.0, 0.0)	Default light source position in the scene.
Light Emission Intensity	(1.0, 1.0, 1.0)	The RGB representation of light emitted radiance.
Maximum Frames Counter	10	The maximum number of frames we keep track of. Given the total number of photons as 200000, the photons emitted for each frame is 20000
Image Resolution	512 x 512	The resolution of result image.
Samples per Pixel	1	Multi-sampling is not used in the experiments thus there is only 1 ray being traced per pixel.
Trace Shadow Rays	Off	Toggle tracing the shadow rays.
Specular Reflection	Off	Toggle of specular reflection when tracing photons.

Chapter 5

Conclusions

In this research, we mainly focus on developing an improved photon mapping technique specifically for current generation GPU taking the advantage of the massive parallelized computation power. We started with introducing the theoretic basis of global illumination, then we reviewed the existing approaches published on the solving the problem, especially the photon mapping techniques exploited the parallelism with GPU. In our opinion, the existing method suffers from deficiencies in terms of how it handles the dynamic scene. After analyzing the weakness, we present a new approach trying to avoid the secondary kd-tree construction exclusively for photon data and instead support an incremental updating scheme by associating the photons data with the kd-tree of the geometric objects and accumulating the photons from previous frames for K Nearest Neighbor photon searching. To proof our concept we implemented the test programs for both the existing and our new approach and carried out a series of benchmarks.

The major contribution of this thesis can be summarized as follows:

- Proposed a new data structure that maintains the kd-tree leaf nodes data and photons data for GPU-based photon mapping renderer and implemented the parallel construction and incremental updating algorithm on this data structure allowing us to efficiently perform radiance estimation using photons.

- Improved the rendering performance for the scenes with dynamic light source by using the new data structure to avoid the overhead brought by re-constructing another kd-tree for photons every frame.

- Add fewer photons to scene per frame with the same image quality compared to the standard photon mapping technique to achieve efficient rendering.

In our tests we observed that our approach was faster during construction and almost all the photon search tests among certain range of frames for a dynamic scene. Especially the construction time was greatly reduced compared to the old approach. Along with speed measurements, we also examined the memory consumption of both approaches. Our approach consumes more memory in rendering phase since the photon data from previous frames. But the memory consumption in the construction phase is much less than the existing method. Because we only require the construction of kd-tree for the scene. The memory consumption also strongly depends on the number of frames we want to store for the photons. Finally, we validate the quality of the rendered by directly visualizing the photons in the scene.

5.1 Future Work

We believe that there are a couple of interesting topics the future work can work on to improve our new approach. The first one is improve the algorithm of the classifying the photons to the kd-tree's leaf nodes to achieve more efficient parallelization. The current solution is that each thread works on one leaf node iterating over all the photons, this could lead considerable deficiency of parallelization on GPU especially when the height of the tree is relative low (there are less leaf nodes) due to the low occupancy of CUDA threads. One possible solution is to map the photons to CUDA threads instead, marking the photons that belong to certain kd-tree leaf node and applying a parallel primitive algorithm such as compact to calculate the indices of photons for each kd-tree leaf node in parallel.

Another interesting application we can explore further is to test and observe that how participating media such as smoke, dust or clouds effect the performance using our approach. The participating media will affect light when it travels through them, the light beam is either absorbed or scattered. Since our approach spatially encodes the distribution of the photons based on the volume boundary of the kd-tree nodes, we think our approach is suitable for storing photon information of volumetric participating media and good performance could be expected.

In addition to keep evolving the approach algorithmically, we certainly cannot ignore the impact brought by the developments in graphics hardware on the photon mapping techniques. With the latest generation of GPUs and development framework, the applications will benefit from better GPU performance and more flexibility, such as better performance for non-coalesced memory access implicit hardware optimizations which is almost free for developers.

Bibliography

[1] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, September 1975.

[2] Johannes Günther, Ingo Wald, and Philipp Slusallek. Realtime caustics using distributed photon mapping. In *Rendering Techniques'04*, pages 111–122, 2004.

[3] Cindy M Goral, Kenneth E Torrance, Donald P Greenberg, and Bennett Battaile. Modeling the interaction of light between diffuse surfaces. *SIGGRAPH Comput. Graph.*, 18(3):213–222, January 1984.

[4] Pat Hanrahan, David Salzman, and Larry Aupperle. A rapid hierarchical radiosity algorithm. *SIGGRAPH Comput. Graph.*, 25(4):197–206, July 1991. ISSN 0097-8930.

[5] J. M. Hasenfratz, M. Lapierre, N. Holzschuch, and F. Sillion. A Survey of Real-time Soft Shadows Algorithms. *Computer Graphics Forum*, 22(4):753–774, December 2003.

[6] Vlastimil Havran. *Heuristic Ray Shooting Algorithms.* Ph.d. thesis, Department

of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, November 2000.

[7] Henrik Wann Jensen. *Realistic Image Synthesis Using Photon Mapping.* 2004.

[8] Nicolas Holzschuch, François Sillion, and George Drettakis. An efficient progressive refinement strategy for hierarchical radiosity. In *In Fifth Eurographics Workshop on Rendering*, pages 343–357. Springer, 1994.

[9] Daniel Reiter Horn, Jeremy Sugerman, Mike Houston, and Pat Hanrahan. Interactive k-d tree gpu raytracing. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, I3D '07, pages 167–174, New York, NY, USA, 2007. ACM.

[10] James T Kajiya. The rendering equation. In *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '86, pages 143–150, New York, NY, USA, 1986. ACM.

[11] Eric P. Lafortune and Yves D. Willems. Bi-directional path tracing. In *Proceedings of third international conference on computational graphics and visualization techniques (compugraphics Š93)*, pages 145–153, 1993.

[12] C. Lauterbach, M. Garl, S. Sengupta, D. Luebke, and D. Manocha. Fast bvh construction on gpus. In *In Proc. Eurographics Š09*, 2009.

[13] D.T. Lee and C.K. Wong. Worst-case analysis for region and partial region

searches in multidimensional binary search trees and balanced quad trees. *Acta Informatica*, 9:23–29, 1977.

[14] D. Luebke and S. Parker. Interactive ray tracing with cuda. *NVISION08,*, 2008.

[15] J. David MacDonald and Kellogg S. Booth. Heuristics for ray tracing using space subdivision. *The Visual Computer*, 6:153–166, 1990. 10.1007/BF01911006.

[16] Shubhabrata Sengupta Mark Harris and John D. Owens. Parallel prefix sum (scan) with cuda. *GPU Gems 3*, 2007.

[17] Ivan Neulander. Adaptive importance sampling for multi-ray gathering. In *ACM SIGGRAPH 2011 Talks*, SIGGRAPH '11, pages 38:1–38:1, New York, NY, USA, 2011. ACM.

[18] nVidia Corporation. Nvidia cuda programming guide. 2012.

[19] Matt Pharr and Greg Humphreys. *Physically Based Rendering, Second Edition: From Theory To Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2010.

[20] Stefan Popov, Johannes Günther, Hans-Peter Seidel, and Philipp Slusallek. Experiences with streaming construction of SAH KD-trees. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pages 89–94, September 2006.

[21] Stefan Popov, Johannes Günther, Hans-Peter Seidel, and Philipp Slusallek. Stackless KD-Tree Traversal for High Performance GPU Ray Tracing. *Computer Graphics Forum*, 26(3):415–424, September 2007. ISSN 0167-7055.

[22] Franco P. Preparata and Michael I. Shamos. *Computational geometry: an introduction*. Springer-Verlag New York, Inc., New York, NY, USA, 1985.

[23] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. *ACM Trans. Graph.*, 21(3):703–712, July 2002.

[24] Timothy J. Purcell, Craig Donner, Mike Cammarano, Henrik Wann Jensen, and Pat Hanrahan. Photon mapping on programmable graphics hardware. In *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05. ACM, 2005.

[25] Maxim Shevtsov, Alexei Soupikov, and Er Kapustin. Highly parallel fast kd-tree construction for interactive ray tracing of dynamic scenes. *Computer Graphics Forum*, 26:395–404, 2007.

[26] Ingo Wald. *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Computer Graphics Group, Saarland University, 2004.

[27] Ingo Wald, Thomas Kollig, Carsten Benthin, Alexander Keller, and Philipp Slusallek. Interactive global illumination using fast ray tracing. In *Proceedings of the 13th Eurographics workshop on Rendering*, EGRW '02, pages 15–24, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[28] Ingo Wald, Johannes Günther, and Philipp Slusallek. Balancing considered harmful – faster photon mapping using the voxel volume heuristic. *Computer Graphics Forum*, 22(3):595–603, 2004. (Proceedings of Eurographics).

[29] Turner Whitted. An improved illumination model for shaded display. *Communications of the ACM*, 23:343–349, 1980.

[30] Kun Zhou, Q Hou, and R Wang. Real-time kd-tree construction on graphics hardware. *ACM Transactions on Graphics (TOG)*, 2008.