M.S. THESIS

Fast and Lightweight Path Guiding Algorithm on GPU

GPU 상에서의 빠르고 가벼운 Path Guiding 알고리즘

BY

Juhyeon Kim

FEBRUARY 2022

DEPARTMENT OF ELECTRICAL ENGINEERING AND
COMPUTER SCIENCE
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

M.S. THESIS

Fast and Lightweight Path Guiding Algorithm on GPU

GPU 상에서의 빠르고 가벼운 Path Guiding 알고리즘

BY

Juhyeon Kim

FEBRUARY 2022

DEPARTMENT OF ELECTRICAL ENGINEERING AND
COMPUTER SCIENCE
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Fast and Lightweight Path Guiding Algorithm on GPU

GPU 상에서의 빠르고 가벼운 Path Guiding 알고리즘

지도교수 Young Min Kim

이 논문을 공학석사 학위논문으로 제출함

2022 년 2 월

서울대학교 대학원

전기 정보 공학부

Juhyeon Kim

Juhyeon Kim의 공학석사 학위논문을 인준함

2022 년 2 월

위 원 장		(서명)
부위원장		(서명)
위 원		(서명)

Abstract

We propose a simple, yet practical path guiding algorithm that runs on GPU. Path guiding renders photo-realistic images by simulating the iterative bounces of rays, which are sampled from the radiance distribution. The radiance distribution is often learned by serially updating the hierarchical data structure to represent complex scene geometry, which is not easily implemented with GPU. In contrast, we employ a regular data structure and allow fast updates by processing a significant number of rays with GPU. We further increase the efficiency of radiance learning by employing SARSA [22] used in reinforcement learning. SARSA does not include aggregation of incident radiance from all directions nor storing all of the previous paths. The learned distribution is then importance-sampled with an optimized rejection sampling, which adapts the current surface normal to reflect finer geometry than the grid resolution. All of the algorithms have been implemented on GPU using megakernal architecture with NVIDIA OptiX [19]. Through numerous experiments on complex scenes, we demonstrate that our proposed path guiding algorithm works efficiently on GPU, drastically reducing the number of wasted paths.

Keywords: Path Guiding, Reinforcement Learning, Ray Tracing

Student Number: 2019-27633

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Path tracing is a Monte Carlo method in the computer graphics field that faithfully simulates light transport to synthesize a photo-realistic image. Basically, path tracing synthesizes an image by estimating the *rendering equation* [9] (Equation 3.1) with Monte Carlo integration that simulates the light transport. Given limited resources, how we sample the light transport path significantly affects the quality of the result image. *Path guiding* focuses on efficiently sampling the ray to reduce the variance of estimates during the Monte Carlo integration. The idea is based on *importance sampling*, where the path with higher contribution to the estimation is sampled more frequently.

The goal in path guiding is to iteratively (i) learn high-energy light paths and (ii) efficiently sample according to the learned distribution. Unlike information known before rendering such as BRDF or the light source that does not consider the complex interplay of geometry (Figure 1.1), path guiding learns the radiance distributions embracing the sophisticated geometry and resulting indirect illumination for each setting.

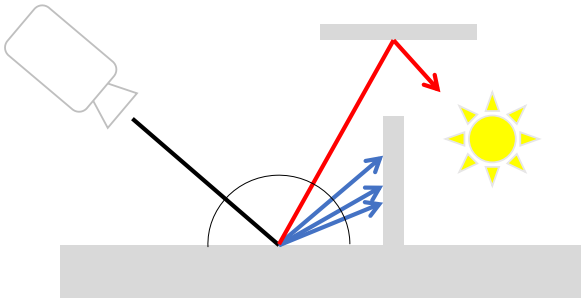

Figure 1.1 The idea of path guiding. Sampling according to BRDF or direct illumination (blue lines) makes it fails to reach the light source. We have to guide the path to follow the red colored path in order to reach the light source.

Although a considerable amount of research has been conducted on path guiding [8, 7, 24, 14], most of them are implemented on the CPU and the GPU case has been rarely studied except a few cases [4]. With the emergence of GPU ray tracing libraries such as NVIDIA OptiX [19], more commercial ray tracing programs will shift to GPU-based versions to take advantage of massively parallel processing. Nevertheless, it is not straightforward to apply path guiding on GPU and even has not been considered in most of the previous works.

In this thesis, we propose a fast and lightweight path guiding algorithm that can be incorporated into the existing GPU path tracing pipeline. We adopt a GPU-friendly data structure that is regular in both spatial and directional domains and store the radiance distribution from previous samples. In order to achieve (i), we exploit knowledge from reinforcement learning (RL). The outgoing radiance can be represented as the weighted sum of incoming radiance, whose online estimation is identified to be similar to the Q value learning in standard RL [2]. Inspired by RL techniques, we propose to use a light algorithm

based on SARSA, which does not require accumulated sum, instead of expected-SARSA [22] that was originally proposed in [2]. As a result, we can achieve fast on-line learning of the radiance distribution in GPU. Meanwhile, regarding (ii), we propose a fast importance sampling with rejection sampling using only half of the directional distribution that overlaps with the normal-oriented hemisphere. A naïve implementation of rejection sampling on the locally-adaptive hemisphere can increase the time complexity as many samples can be rejected and we need to calculate the adaptive normalization factor. The ratio of rejection can be controlled by mixing the sampling distribution with the uniform distribution, and the online normalization can be accelerated with memoization. The sampling algorithm can be efficiently implemented under a GPU environment.

To sum up, the major contributions of the thesis could be listed as follows.

- We propose a path guiding algorithm that employs regular spatial-directional data structure and light-weight computation such that we can highly utilize GPU and accelerate the entire process.
- Inspired from the analogy between RL and light transport [2], we introduce a lightweight RL algorithm (SARSA) that can efficiently update radiance without aggregating all previous samples.
- We suggest a fast importance sampling using the rejection sampling method that adapts the learned directional distribution to the local geometry with an optimized mixture distribution.

We implement the data structure and sampling algorithm on GPU and demonstrate that the proposed method can accelerate the path-guiding algorithm in various scenes.

Chapter 2

Background and Related Works

2.1 Ray Tracing on GPU

The recent advance of GPU technology has enabled ray tracing to be implemented in the GPU environment. The major difference between CPU and GPU environments is the number of concurrent threads. Unlike CPU which only allows a handful of threads, GPU allows thousands of threads to run simultaneously with SIMT (Single Instruction Multiple Threads) execution models. The algorithm must be structured to allow thousands of threads to concurrently operate, read/write memory, and perform instructions, with careful memory management.

NVIDIA's OptiX [19] is one of the first user-programmable GPU ray tracing engines that fully exploits the aforementioned nature of GPU. It is implemented in a single *megakernel* architecture that encompasses all components for ray tracing - ray generation, ray intersection, and material codes. [12] pointed out that megakernel suffers from divergence issue under SIMT execution, and pro-

posed wavefront-based rendering that propagates a large set of rays together using large memory. Each approach has advantages and disadvantages, but recent GPU architecture is designed highly optimized for megakernel [18]. Therefore, we decided to use megakernel for our work, but we also designed it to work as efficiently as possible on wavefront architectures.

2.2 Path Guiding

Path guiding aims to learn indirect radiance and utilize it for efficient importance sampling. Earlier works proposed different data structures to store the directional distribution on the surface points such as photon map [8], rasterized 5D tree [11] or hemispherical particle footprints [7]. However, these works precalculate the radiance offline which is approximated from a limited number of initial particles. Also, it is not known how many samples need to be processed for the approximation to converge to a decent distribution.

To overcome the limitation of offline path guiding, recent works proposed to update the distribution online while performing the path tracing. Several works [6, 23, 24] proposed to represent the sampling distribution using Gaussian Mixture Model (GMM) and used EM algorithm to mix the offline and online learning. The online distribution can also be efficiently updated using a hierarchical data structure. Müller et al. [14] proposed using the binary tree for the spatial distribution and quadtree for the directional distribution (SD-tree). The algorithm can be further optimized to consider the stochastic behavior of the ray samples [13, 20] or use the linear cosine transform [3] for product importance sampling.

All of the previous path guiding algorithms with online updates adaptively store the distribution, and the ray tracing has to be processed in the CPU.

The radiance distribution can also be stored and sampled using a neural network [16, 15], where the neural network is implemented in GPU but the ray tracing is left to CPU. Our algorithm is one of the first path-guiding frameworks to run the light transport in parallel using GPU with online updates of the radiance distribution. Note that Dittebrandt et al. [4] recently proposed real-time GPU path guiding based on simplified SD-tree. They focused on the real-time performance using only one sample per pixel and ignoring multiple bounces of paths, and the quality of rendered images is inevitably lower than what we are targeting.

2.3 Reinforcement Learning and Light Transport

Another interesting line of works [2, 10] observed that the rendering equation and expected-SARSA [21], a well-known variation of Q-learning, have structural similarity. The equation for the online updates of the value function in reinforcement learning resembles that for the online radiance updates, which provided the theoretical inspiration to our work. However, expected-SARSA requires summing over all possible actions which is computationally heavy. Also, their proposed method requires sequential updates and cannot be directly applied to concurrently run in a GPU.

Chapter 3

Problem Setting and Overview

The rendering equation [9] describes the outgoing radiance $L_o(x, \omega_o)$ from point x toward outgoing direction ω_o as

$$L_o(x, \omega_o) = L_e(x, \omega_o) + \int_\Omega L_i(x, \omega_i) f_r(x, \omega_o, \omega_i)(n \cdot \omega_i) \mathrm{d}\omega_i. \tag{3.1}$$

The first term $L_e(x, \omega_o)$ is the emitting radiance from the location x in the direction of ω_o and the second term aggregates the reflection of incoming radiance. Specifically, the incident radiance $L_i(x, \omega_i)$ from the direction ω_i is attenuated by $n \cdot \omega_i$, the cosine angle between the surface normal and the incident angle. The proportion reflected into the outgoing direction ω_o is defined with the bidirectional reflectance distribution function (BRDF) $f_r(x, \omega_o, \omega_i)$. The product of the three terms L_i, f_r, and $(n \cdot \omega_i)$ is integrated over all of the incoming direction in normal-oriented hemisphere Ω.

Path tracing evaluates the above rendering equation using Monte Carlo

integration with N samples

$$\langle L_o(x,\omega_o)\rangle = L_e(x,\omega_o) + \frac{1}{N}\sum_{j=1}^{N}\frac{L_i(x,\omega_j)f_r(x,\omega_o,\omega_j)(n\cdot\omega_j)}{p(\omega_j|x,\omega_o)}, \tag{3.2}$$

where $p(\omega_j|x,\omega_o)$ is the sampling PDF for the incoming direction ω_j given the position x and the outgoing direction ω_o. The variance of the estimator in Equation 3.2 depends on the sampling PDF $p(\omega_j|x,\omega_o)$. Ideally, zero-variance could be achieved if the sampling PDF is exactly proportional to the numerator. Out of the three terms in the numerator, BRDF f_r and cosine term $n\cdot\omega_j$ can be easily determined but incident radiance term $L_i(x,\omega_j)$ remains unknown before rendering. However, within a complex scene where the illumination condition changes according to the local geometry, it is crucial to consider the radiance distribution $L_i(x,\omega_j)$.

The incident radiance field $L_i(x,\omega)$ is a function of position x and direction ω and the distribution is represented with 5D data structure. $L_i(x,\omega)$ is highly irregular due to the interplay between the geometric configuration and light, and needs to be approximated with samples of rays using Equation 3.2. The complex distribution is commonly represented with a hierarchical data structure, which is updated online as more samples are processed. The frequent update of the adaptive data structure cannot be easily implemented with GPU.

To make the path guiding available in GPU, we adapt the framework to be parallelizable by employing GPU-friendly regular data structure and concurrent updates of the distribution. As illustrated in Figure 3.1, the scene is divided into voxels, and the directions are discretized using the equal-area projection with regular shape as proposed in [5]. We learn the radiance field and store it in the aforementioned grid-based structure with a fast and lightweight RL algorithm called SARSA to circumvent computationally heavy radiance estimation as described in Chapter 4. We also suggest a fast rejection sampling method

to quickly advocate for the fine geometry within the grid cell as presented in Chapter 5. All of the implementations are built on megakernel architecture to fully utilize the computation capability of the modern GPU.

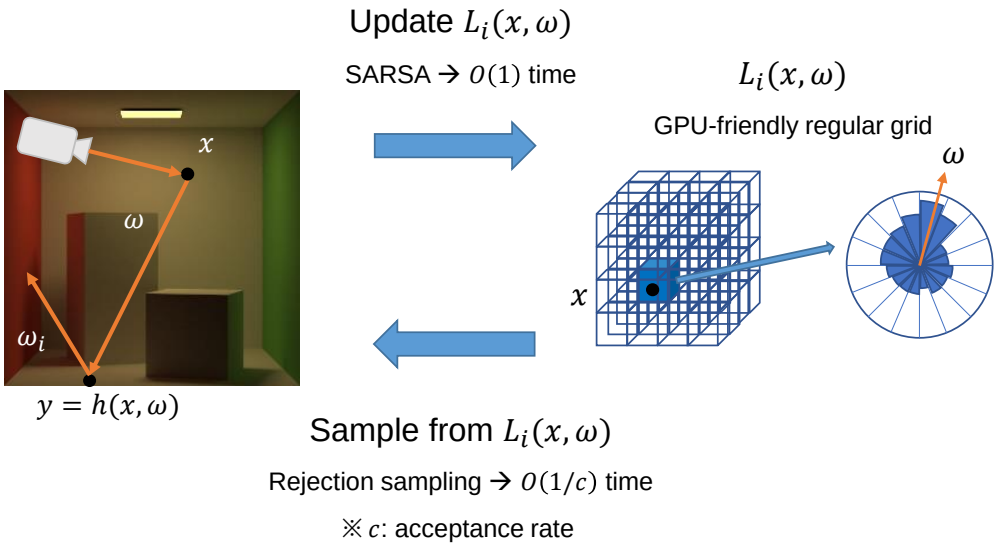

Figure 3.1 The overall flow of our proposed method. We store incident radiance $L_i(x,\omega)$ in GPU-friendly regular data structure. We process rays and efficiently update $L_i(x,\omega)$ with SARSA, while, from $L_i(x,\omega)$, we quickly sample valid rays to render the scene with rejection sampling.

Chapter 4

Fast and Lightweight Radiance Learning

We propose a fast and lightweight method to learn the complex distribution of the radiance taking inspiration from the frameworks in reinforcement learning (RL).

4.1 Analogy between the Rendering Equation and Reinforcement Learning

As Dahm et al. [2] had identified, rendering equation has a close relationship with RL. Like the action-value function in RL, the rendering equation in Equation 3.1 can be written in a recursive way. If there is no participating media, the incoming radiance $L_i(x, \omega)$ is the same as the outgoing radiance $L_o(y, -\omega)$, where $y = h(x, \omega)$ is the hitpoint when we trace the ray from x to direction ω. Using this representation, we can rewrite the rendering equation in a recurrent

Table 4.1 Analogy between RL (Eq. 4.2) and rendering equation (Eq. 4.1)

Reinforcement Learning (Eq. 4.2)	Rendering Equation (Eq. 4.1)
$s, s' \in S$	$x, y \in \mathbb{R}^3$
$a, a' \in A$	$\omega, \omega_i \in \Omega$
$Q(s, a)$	$L_i(x, \omega)$
$r(s, a, s')$	$L_e(y, -\omega)$
$Q(s', a')$	$L_i(y, \omega_i)$
$\pi(s', a')$	$f_r(y, -\omega, \omega_i)(n \cdot \omega_i)$

form using the incident radiance L_i:

$$\begin{aligned} L_i(x, \omega) &= L_o(y, -\omega) \\ &= L_e(y, -\omega) + \int_\Omega L_i(y, \omega_i) f_r(y, -\omega, \omega_i)(n \cdot \omega_i) \mathrm{d}\omega_i. \end{aligned} \tag{4.1}$$

While coming from a different context, the recursive equation in Equation 4.1 resembles the equations in RL. Given a set of states S and a set of actions A, an agent at state s takes an action a and transit to the next state s' receiving the reward $r(s, a, s')$. For the sake of finding the optimal policy of actions, we often define the action-value function $Q(s, a)$ as an expected cumulative reward, and iteratively update it. Expected-SARSA, one of the algorithms to update the Q function, acts with the following equation:

$$\begin{aligned} Q(s, a) \leftarrow &(1 - \alpha) \cdot Q(s, a) \\ &+ \alpha \cdot \left(r(s, a, s') + \gamma \int_A \pi(s', a') Q(s', a') \mathrm{d}a' \right), \end{aligned} \tag{4.2}$$

where $\pi(s', a')$ (also known as the *policy*) is a probability to choose next action a' at the next state s', α is a learning rate and γ is a discount factor.

Because of structural similarity between Equation 4.1 and the update target in Equation 4.2 which is summarized in Table 4.1, learning radiance distribu-

tion $L_i(x, \omega)$ can be achieved using expected-SARSA with $\gamma = 1$ [2]. Regarding the integral term, [2] used stratified sampling over the hemisphere. More specifically, the radiance $L_i(x^{(m)}, \omega^{(m)})$ at m-th iteration of length M path ($m < M$) (Figure 4.1) can be updated as following

$$\begin{aligned} & L_e(x^{(m+1)}, -\omega^{(m)}) \\ & \quad + \frac{2\pi}{N} \sum_{k=1}^{N} L_i(x^{(m+1)}, \omega_k) f_r(x^{(m+1)}, -\omega^{(m)}, \omega_k)(n \cdot \omega_k), \end{aligned} \tag{4.3}$$

where $x^{(m+1)} = h(x^{(m)}, \omega^{(m)})$. The sampling direction of the hemisphere is equally partitioned into N stratum and ω_k for each stratum is extracted by stratified sampling with uniform probability $p(\omega_k|x, \omega) = 1/2\pi$. This method is computationally heavy because it includes aggregating N incident radiance which involves the BRDF computation $O(N)$ times.

4.2 Fast and Lightweight Radiance Learning with SARSA

We examine algorithms to update the Q function in RL that could be used in the place of expected-SARSA for the fast and light-weight update. Basically, in RL, prediction that estimates Q function, could be largely categorized into three groups; dynamic programming (DP), Monte Carlo (MC), and temporal difference (TD) methods. Figure 4.1 illustrates how the aforementioned Q value prediction algorithms can be interpreted in radiance learning.

DP updates Q value to expected future return that is aggregated over whole next possible actions with full-width backups. The radiance update using the expected-SARSA in Equation 4.3 could be regarded as DP method. It involves the exhaustive aggregation over the hemisphere as DP, also indicated in Figure 4.1(a). While DP can lead to more accurate estimation with low variance and low bias, it is computationally heavy and considerably slower than MC or

TD.

MC method updates Q value to the actual return from complete episodes without bootstrapping. When it is applied for the radiance update (Figure 4.1(b)), one has to store all previous bounces $(x^{(j)}, \omega^{(j)})$ and update the radiance of the entire sequence when the ray finally reaches the light source

$$\sum_{j=m+1}^{M} \left(L_e(x^{(j)}, -\omega^{(j-1)}) \prod_{k=m+1}^{j-1} a^{(k)} \right), \tag{4.4}$$

where $a^{(j)}$ is the attenuation factor which is defined as follow:

$$a^{(j)} = \frac{f_r(x^{(j)}, \omega^{(j-1)}, \omega^{(j)})(n \cdot \omega^{(j)})}{p(\omega^{(j)}|x^{(j)}, \omega^{(j-1)})}. \tag{4.5}$$

Having a full path to the terminate state, MC tends to have low bias. However using distinct paths instead of bootstrapping results in high variance. Furthermore, more importantly, MC requires a large memory because we have to store all intermediate points. It can be a serious problem when we concurrently process a large number of rays. For example, a wavefront-based rendering typically maintains a pool of millions of rays and storing the length M paths for all rays would be disastrous.

TD method updates Q value to expected future return with bootstrapping. The only appropriate variation of TD method is SARSA update

$$Q(s,a) \leftarrow (1-\alpha) \cdot Q(s,a) + \alpha \cdot \left(r(s,a,s') + \gamma Q(s',a') \right). \tag{4.6}$$

Note Q-learning, the other TD method, only considers the maximum Q value and cannot accurately estimate the integral over incident radiance [2]. The SARSA's update target is similar to expected-SARSA, while it only differs in estimating future expected rewards

$$L_e(x^{(m+1)}, -\omega^{(m)}) + a^{(m+1)} L_i(x^{(m+1)}, \omega^{(m+1)}). \tag{4.7}$$

Unlike expected-SARSA, SARSA only considers the single next action, which does not require aggregation (Figure 4.1(c)). Therefore, time complexity could be reduced from $O(N)$ to $O(1)$ where N is a number of the possible actions. Although MC also takes $O(1)$ time to update a single path, it empirically turned out to take more time compared to SARSA, which may be due to read/write overhead from the array that stores previous points. In conclusion, we suggest to use SARSA for radiance learning because of its superiority in speed and memory consumption compared to MC or expected-SARSA, which is further verified in the experiments.

The update in Equation 4.7 is attenuated by α and combined with Equation 4.6, yielding the full update equation

$$L_i(x, \omega) \leftarrow (1 - \alpha) \cdot L_i(x, \omega) + \alpha \left(L_e(y, -\omega) + a_y L_i(y, \omega_i) \right) \tag{4.8}$$

where a_y is the attenuation factor at $(y, -\omega, \omega_i)$. However such update still suffers from the concurrency issue, and possibly yields the race condition in GPU environment. We resolve the problem by separating the rendering iteration into a few steps and updating L_i in a batch.

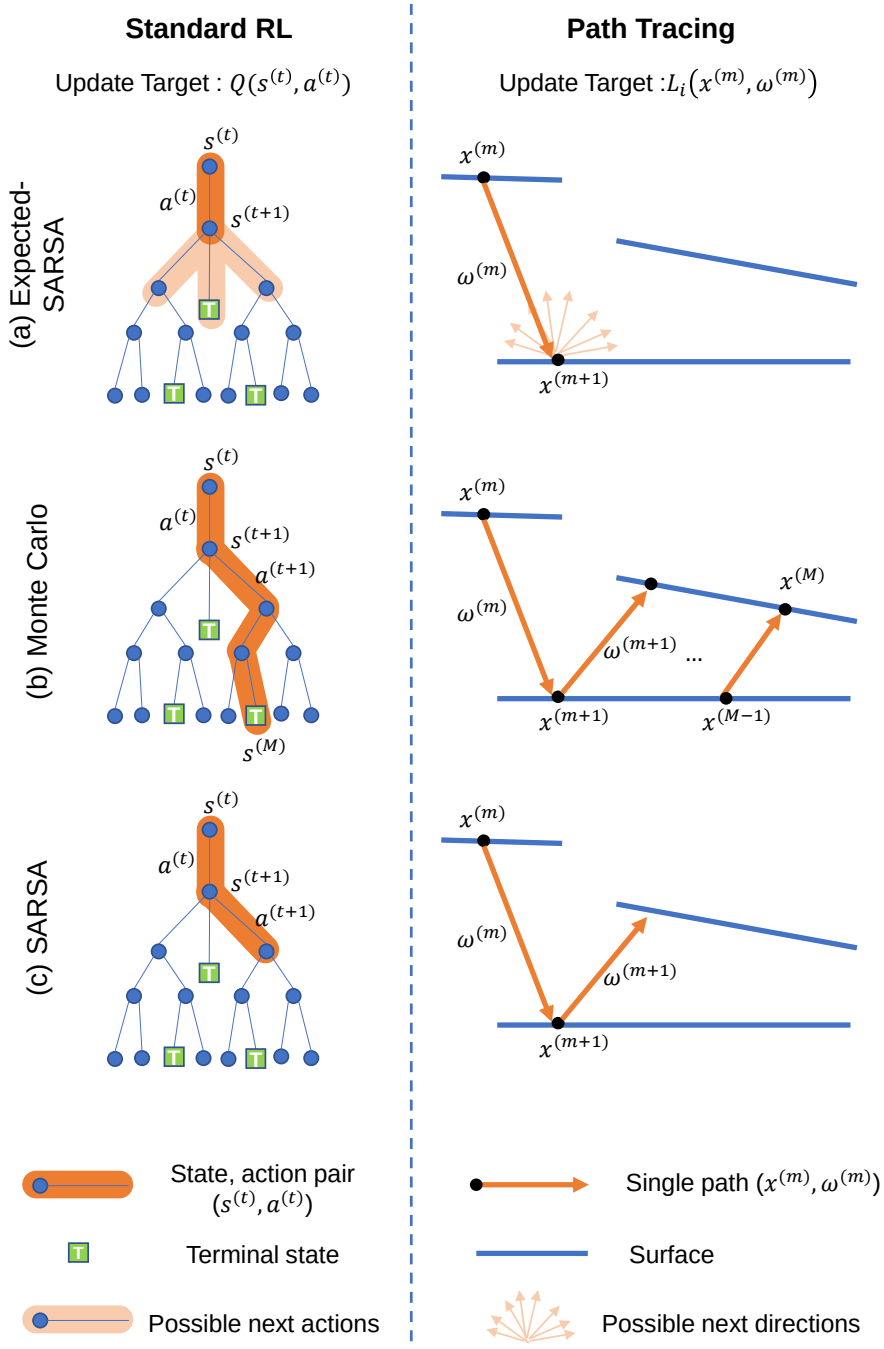

Figure 4.1 Difference between three updating method (a) expected-SARSA, (b) Monte Carlo and (c) SARSA in standard RL and path tracing.

Chapter 5

Efficient Importance Sampling from Learned Radiance

As we update the radiance value, we concurrently run path guiding in Equation 3.2 using the estimated distribution as the sampling distribution $\omega \sim p(\omega_i|x,\omega_o) \propto L_i(x,\omega_i)f_r(x,\omega_o,\omega_i)(n\cdot\omega_i)$ where the value of the product is approximated for each of the distcretized angle. Note that the distribution jointly considers the radiance, BRDF, and the cosine term such that we can effectively perform product important sampling. The ray samples are quickly processed in GPU with an efficient hemisphere sampling and memoization technique to substitute computing the marginalization of the online PDF.

5.1 Importance Sampling on Hemispherical Domain

The 5D radiance field $L_i(x,\omega)$ is tabulated as a coarse spatial grid, and within each 3D cell, the 2D directional radiance distribution is stored in a spherical domain as shown in Figure 5.1. The direction sampling methods in path guiding

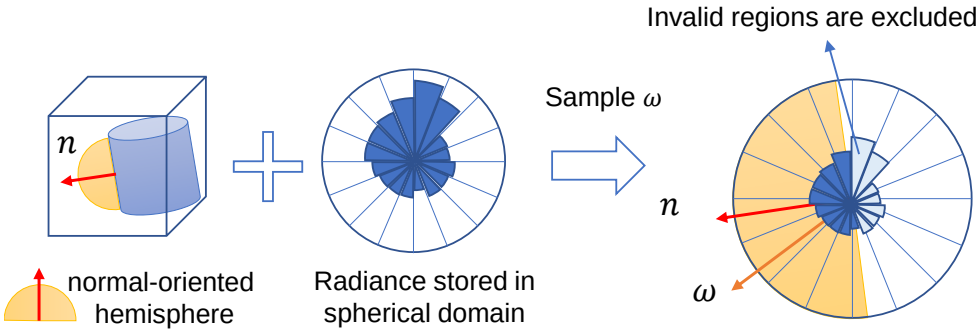

Figure 5.1 The hemispherical domain sampling removes probability of sampling from invalid hemisphere.

can either consider the entire sphere or only a hemisphere that aligns with the surface normal [2, 24]. While the hemisphere distribution avoids sampling the invalid directions that face inside the surface, the individual spatial grid cells that contain different normal directions have to be in fine resolution such that correctly reflect the complex scene geometry. The hemisphere domain requires a prohibitive amount of memory in complex scenes, for example, human hair or tree leaves as demonstrated by [14]. In contrast, our approach considers the full spherical directions and does not need to correctly represent the surface normal directions. It does not have the memory requirement as hemisphere distribution and can be implemented with much lower memory in GPU even in complicated scenes. However, since sampling in spherical domain is not normal-sensitive, it is likely that many invalid samples that face inside the surface are produced.

We propose a hybrid approach to efficiently sample the distribution considering the local geometry. Specifically, we find the intersection between the stored spherical distribution and the hemisphere that aligns with the normal

direction of the current surface point, and sample only from the intersection distribution as illustrated in Figure 5.1. While our GPU-friendly grid structure may suffer from lower directional resolution than the quadtree-based implementation [14], our regularity can quickly adapt to finer geometry within the cell and therefore efficiently utilized to product importance sampling. The sampled distribution $p(\omega)$ changes according to the normal direction of the hitpoint and minimizes wasted samples.

5.2 Fast and Efficient Importance Sampling with Optimized Rejection Sampling

We propose to perform importance sampling from learned radiance with optimized rejection sampling. In fact, the evolving distribution $p(\omega)$ can be sampled in three possible ways including our proposed method. We will review each sampling strategy and discuss its advantage and disadvantage.

Inversion sampling The most intuitive way is *inversion sampling* method, which samples from the cumulative density function (CDF). For a static distribution, the CDF can be pre-computed and the inverse sampling can be executed in $O(\log N)$ with a binary search. However, we are sampling from the intersection of the normal-oriented hemisphere and the spherical distribution, and we have to dynamically construct the CDF for each direction which requires $O(N)$ time complexity.

Rejection sampling Another possible way is *rejection sampling*, which is also known as the dart-throwing approach. Rejection sampling first produces sample from other easy-to-sample PDF $u(\omega)$ such that $cp(\omega) < u(\omega)$ for some scalar value c. Then, it probabilistically accepts the sample only if $\eta < cp(\omega)/u(\omega)$ for a uniform random variable η. If u is a uniform distribution and c is set to

the tightest value ($1/p_{\max}$), then c becomes the same as the acceptance rate which represents the relative area between the two distributions, as illustrated in Figure 5.2, left. High c implies less rejection. In our setting, we use the uniform distribution $u(\omega)$ whose domain is the normal-oriented hemisphere.

One possible drawback of the rejection sampling is rejecting too many samples if there is a large discrepancy between the distribution $u(\omega)$ for the initial sampling and the true distribution $p(\omega)$ used for rejection. For example, if the sampling distribution is concentrated at a narrow range of ω, then most of the uniform samples can be rejected, hurting the overall performance. The speed of rejection sampling can be improved with mixed distribution

$$p_{\text{sampling}} = (1 - \epsilon)p + \epsilon u, \tag{5.1}$$

where $\epsilon \in [0, 1]$ is a constant value that controls the trade-off between using the correct distribution for the importance sampling and the high rejection rate. The effect of the mixed distribution is also illustrated in Figure 5.2. When the sample is generated with the modified distribution in Equation 5.1, the acceptance rate for mixed distribution would be $c' = c + (1 - c)\epsilon$ instead of the accept rate c of the original rejection sampling. Since only $(1 - \epsilon)$ of the samples are extracted from p, we can expect $(1 - \epsilon)\,(c + (1 - c)\epsilon)$ is the amount of samples extracted from p. If we know the exact c, we can solve for the optimal ϵ by maximizing the number of effective samples with the following optimization

$$\arg\max_{\epsilon \in [0,1]} (1 - \epsilon)\,(c + (1 - c)\epsilon) \tag{5.2}$$

and the optimal ϵ is $\max(\frac{1-2c}{2-2c}, 0)$. Choosing the appropriate ϵ is important to guarantee the performance of the rejection sampling, which is further evaluated in Section 6.3.

Metropolis sampling *Metropolis sampling* is a kind of Monte Carlo Markov Chain algorithm, where samples are drawn from an arbitrary mutation function

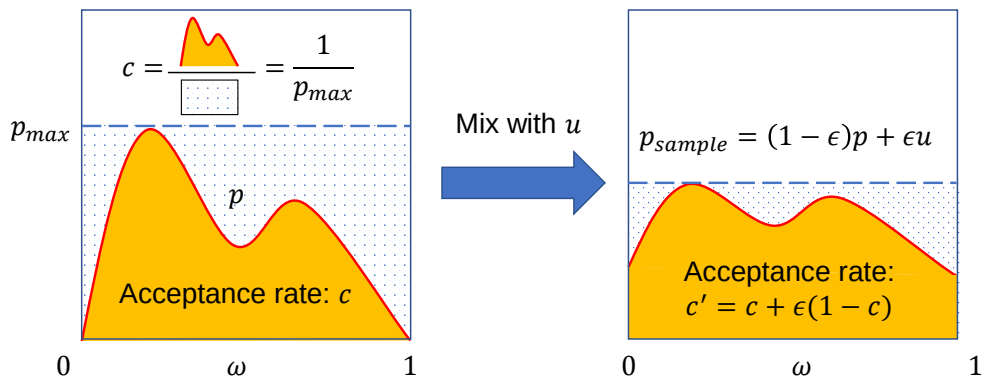

Figure 5.2 Rejection rate alleviation with mixing uniform PDF.

and then the samples are mutated with a pre-defined probability. There can be various mutation functions, but we used simple two strategies, random and adjacent movements with a probability of 0.1 and 0.9, respectively. However, such sequential mutation on a random variable cannot be correctly implemented on GPU, possibly causing race conditions that update the same random variable simultaneously.

Although we introduced 3 possible methods for importance sampling, we suggest using rejection sampling with our proposed optimization strategy since inversion sampling and Metropolis sampling have disadvantages of speed and race condition respectively.

5.3 Normalizing Term Calculation with Memoization

In our implementation, we further accelerate the pipeline on GPU with *memoization* for the normalizing constant. Note that the rejection sampling or Metropolis sampling does not require normalized distribution, and we sam-

ple from the available un-normalized distribution $L_i(x, \omega_i) f_r(x, \omega_o, \omega_i)(n \cdot \omega_i) \propto p(\omega_i | x, \omega_o)$. However, we need to scale the distribution so that its sum becomes one in order to finally evaluate the Monte Carlo integration as described in Equation 3.2, or to find the $p_{\max}$ for the optimized rejection sampling. The normalization term changes frequently because we only consider the hemisphere that aligns with the local surface normal instead of using the stored whole spherical directions. To make the problem simple, we only apply guiding to diffuse-like materials and avoid repetitive calculation with memoization. In this case, we can store the normalizing factor $N(x, \omega_o, n) \simeq N(x, n)$ at position x with normal n by using the same data structure to store the incident radiance field $L_i(x, \omega)$. Similarly, we can store and memoize $p_{\max}$ which is used for mixing distribution in rejection sampling.

Chapter 6

Experiments and Results

We implemented our algorithm on a GPU environment with megakernel architecture. While there are some customizable renderers available based on wavefront architecture such as Mitsuba2 [17], there is no well-established physically-based renderer for megakernel architecture. We wrote the path-guiding algorithm with our own renderer using OptiX [19] and built several BRDFs referring to the rich material library of Mitsuba2[17]. The algorithm is transplanted on Python environment using wrapper PyOptix for faster testing.

We tested our algorithm for 12 scenes from [1] with varying geometries, materials, and complexity. All of the path guiding methods used unidirectional path tracing without *next event estimation* (NEE) for simplicity as [14] and the following works. The reference images were prepared using standard BRDF proportional sampling with 131,072 *samples per pixel* (spp) and maximum depth 32 without Russian roulette. The reference image is used to evaluate the quality of the rendered image by comparing the mean absolute error (MAE). For each path guiding algorithm, a time budget (40 sec) or spp budget (1024) was im-

posed, but a time budget was mainly used for fair comparison. Maximum depth was set to 16 and Russian roulette was set to begin after depth 8. Learning and rendering were fused into the same pipeline in a totally online manner. Instead of exponential growth in [14], we used the constant number of samples per iteration and accumulated the distribution over the iteration. Learned distribution was updated for new distribution at every step that single step is composed of 8 spp. We also forced to sample randomly at the first few steps like the epsilon greedy algorithm in RL, to encourage enough exploration. Spatial and directional resolution was both set to 8, 16 respectively ($8^3 \times 16^2$). We also tested higher resolutions, but we found that too high resolution rather increased the error. Note that the sampled directional grid represents the hemisphere and the amount of stored directional domain size is twice the sampled grid representing a sphere. The details of implementing directional grid are the same with [5].

6.1 GPU-based Path Guiding with a Regular Grid

Our path guiding algorithm using a regular grid is compared against the BRDF-based method and path guiding using the quadtree [14] in Table 6.1 and Figure 6.4, 6.5, 6.6. Quadtree structure adaption is implemented using additional OptiX kernel and updated per exponentially growing steps with flux threshold 0.01. We also used multiple importance sampling with BRDF with a probability of 0.5, the same with the original paper. We set the maximum leaf node number the same as the size of the regular grid so that the total memory used remains unchanged. Also, as the original paper, it is set to use the MC method only to learn radiance.

The BRDF-based method can be easily implemented on GPU and fast, leading to process more number of samples for equal-time comparison. However,

Table 6.1 Equal time comparison for several methods. BRDF-based method samples the ray according to BRDF and does not consider the radiance distribution, and quadtree-based method is our implementation of [14] on GPU. We also show several variations of path guiding using our proposed regular grid structure. 'Ours without Rej+' samples the distribution without rejection optimization. 'Ours without SARSA' utilizes expected-SARSA for radiance learning [2].

Scene Name	BRDF	Quadtree	Ours w/o Rej+	Ours w/o SARSA	Ours
Bathroom	0.0374	**0.0358**	0.0554	0.0387	0.0366
Bathroom-2	0.0345	0.0339	0.0338	0.0344	**0.0288**
Cornell-Box	0.0114	**0.0062**	0.0093	0.0098	0.0072
Cornell-Box-Hard	0.0216	0.0134	0.0182	0.0181	**0.0134**
Kitchen	0.0227	0.0209	0.0198	0.0216	**0.0190**
Living-Room	0.0092	**0.0087**	0.0180	0.0130	0.0116
Living-Room-2	0.0190	0.0181	0.0197	0.0189	**0.0169**
Living-Room-3	0.0558	0.0611	0.0767	0.0622	**0.0511**
Staircase	0.0144	0.0105	0.0164	0.0122	**0.0094**
Staircase-2	0.0146	0.0101	0.0178	0.0107	**0.0092**
Veach-ajar	0.0747	**0.0640**	0.2010	0.0772	0.0745
Veach-ajar-2	0.1233	0.1066	0.1222	0.1323	**0.1029**
Mean (MAE)	0.0366	0.0324	0.0507	0.0374	**0.0317**
Time per Sample (ms)	10.59	11.18	54.73	34.48	16.30
Samples per Pixel	4005	3774	1302	1239	2523
Invalid Sample Rate	0	0.1719	~ 0	~ 0	~ 0

the quality of the produced image does not meet that of path tracing especially when there is complicated occlusion and inter-reflection leading to larger mean absolute error (MAE). The quadtree update is fast enough and also gives better results than pure BRDF sampling, but seems to suffer from invalid samples that head down to the surface. On the other hand, our implementation of the path guiding algorithm is normal-sensitive, thus providing nearly zero invalid samples. It has an advantage over quadtree by product importance sampling with GPU-friendly regular grid structure. Our method evolves to converge to the true radiance distribution L_i and sample more efficient paths as the iteration proceeds. This can be verified by counting the number of rays that hit the light source for each iteration as shown in Figure 6.3-(a). Compared to BRDF sampling, our method achieves $10 \sim 20$ times higher hit rate. Furthermore, we found that our proposed radiance learning method (SARSA) and radiance sampling method (rejection sampling with optimization) both play a pivotal role in performance improvement, which is further discussed in the following sections.

6.2 Comparison for Radiance Learning Methods

In this section, we compare several radiance learning methods discussed in Chapter 4, which are namely based on expected-SARSA, Monte Carlo and SARSA. Table 6.2 shows SARSA is the best choice for radiance learning in GPU, leading to the smallest noise when rendered with an equal time limit. This is mainly due to the fast speed of SARSA. Compared to the BRDF sampling method that does not involve any radiance learning, the computational time of SARSA turns out to be minimal. In contrast, the increase of computation time for expected-SARSA is nearly $\times 2.1$, which significantly decreases the number of

completed samples under the equal time budget. MC is fairly fast, but slightly slower than SARSA which may be due to accessing a record that stores previous points.

Figure 6.1 shows an example of the learned radiance field using the three RL methods. It is widely known in RL that SARSA tends to have higher bias, while Monte Carlo method tends to have higher variance [22]. We can easily verify this in Figure 6.1 that Monte Carlo method results in spotty noise. Expected-SARSA and SARSA are known to be biased, which means they cannot generate the correct reference image even though we the increase number of samples. However, by comparing equal-spp results, we found out that the variance or bias of approximated radiance field have a minimal effect on the final image, and speed is a more important factor when time becomes the budget.

Memory consumption is also an important issue for practical path guiding in GPU. Expected-SARSA and SARSA do not require additional memory. However, Monte Carlo method stores every intermediate point (the maximum could be limited as 32 in [14]) which may require a considerable amount of memory. The approximated memory usage can be calculated by (size of single data) $\times$ (maximum concurrent ray) $\times$ (maximum depth). In our setting, the distribution is stored in a total 12 floats for single data, 16 maximum depth, and about 46,000 concurrent rays, indicating that Monte Carlo method causes an additional 35 MB of stack usage. Of course, this may still be harmful for performance, the more serious problem occurs when we use the wavefront-based method that have to keep millions of rays; it would lead to significant memory usage (1 million $\sim$ 768 MB). Therefore, we could conclude that our SARSA-based update is fast, memory-efficient, while also competent in performance.

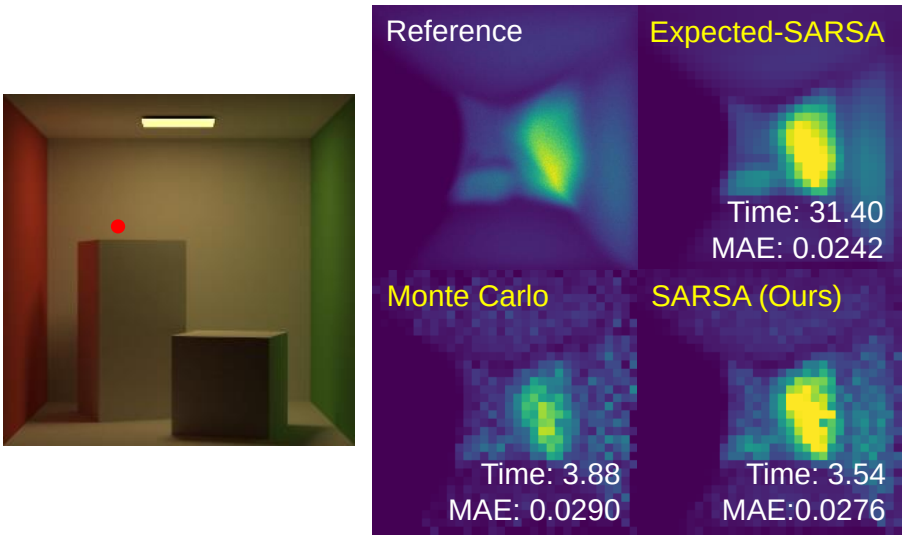

Figure 6.1 The learned radiance map at the position indicated as a red dot in the scene on the left. MAE and required time per sample (ms) are showed. We increased directional grid resolution to emphasize the difference.

6.3 Comparison for Radiance Sampling Methods

In this section, we compare the radiance sampling methods covered in Chapter 5, namely the inversion, rejection, and Metropolis sampling methods. The quantitative result is in Table 6.2 and pseudocode for the implementation could be found in Appendix B.

The simplest way is the inversion method using the stored spherical distribution without considering normal. Ignoring the local geometry, the CDF does not change and can be calculated beforehand with a minimal overhead of $O(\log N)$ where N is the number of directional grid bins. Despite the speed, a significant number of samples are invalid representing rays that direct toward the inside of the surface, resulting in degradation of the quality.

Table 6.2 Equal time comparison for different learning and sampling method discussed in Chapter 4 and Chapter 5. Metropolis sampling is skipped since it turns out to be unstable due to racing condition.

MAE	Sphere	Hemisphere		
	Inv	Inv	Rej	Rej+
Expected -SARSA	0.0425	0.0474	0.0521	0.0374
MC	0.0368	0.0467	0.0524	0.0332
SARSA	0.0340	0.0434	0.0507	**0.0317**

Time per Sample(ms)	Sphere	Hemisphere		
	Inv	Inv	Rej	Rej+
Expected -SARSA	21.62	45.57	66.37	34.48
MC	10.94	26.09	76.61	17.67
SARSA	9.23	25.37	54.73	16.30

We can overcome the limitation by considering the valid hemisphere that aligns with the surface normal. Overall, our proposed rejection-based sampling with optimization gave the best result. The inversion method with the hemisphere sampling involves calculating the normal-adaptive CDF online, which is $O(N)$, and it is no longer fast.

Rejection sampling can be an alternative method because theoretically the time complexity is $O(1/c)$ where c is the acceptance rate. With a naïve implementation, however, the rejection sampling does not improve the performance. A significant number of samples is rejected due to the discrepancy between the initial and the target sampling distribution. We can achieve faster sampling by optimizing ϵ that mixes the distributions as described in Equation 5.1. The optimized rejection sampling (indicated with post-fixed '+' sign in Table 6.1 and 6.2) results in the best quality image for the equal time comparison, greatly reducing the time. The sampling complexity of the optimized version is $O(1/c')$ where $c' = c + (1 - c)\epsilon$ is an acceptance rate for the mixed PDF as proposed in Section 5.2.

Note that Metropolis sampling is lightweight with the time complexity of $O(1)$, but the race condition appears to be causing the crucial performance degradation. Figure 6.2 shows exemplar patches with noticeable artifact.

Effect of ϵ in Equation 5.1 We further investigate the effect of mixing the sampling distributions with different $\epsilon \in [0.0001, 1]$ in Figure 6.3. Figure 6.3-(b) confirms that the performance of SARSA (TD) is better than BRDF-based sampling or other RL-based algorithms such as expected-SARSA (DP) or Monte Carlo when implemented in GPU. The optimal ϵ allows us to efficiently sample the rays, and clearly leads to performance improvement. Figure 6.3-(c) further scrutinize the effect of different ϵ with SARSA. With a small ϵ, we could draw more samples proportional to radiance such that the hit rate increases, but too

Figure 6.2 Metropolis sampling causes visual artifact.

many samples get rejected which drastically increases time to sample and reduces the number of samples. As we increase ϵ, while it increases the acceptance rate, the rejection optimization dilutes the estimated radiance distribution L_i. As a result, we can observe that the hit rate doubles without the rejection optimization (Figure 6.3-(a)). The optimal value has to balance between the number of samples and the hit rate, and we found the minimum MAE for ϵ near 0.5.

Effect of memoization Another key factor that speeds up the rejection sampling is memoization, which pre-calculates the normalizing factor. However, the pre-calculated values might be inaccurate due to the recurrent updates of distribution during the Monte Carlo integration. We compared images produced with the same number of samples using our sampling approach and compared the MAE and the runtime. Empirically we observed that the inaccuracy is negligible while the reduction in the total calculation time is about 15%.

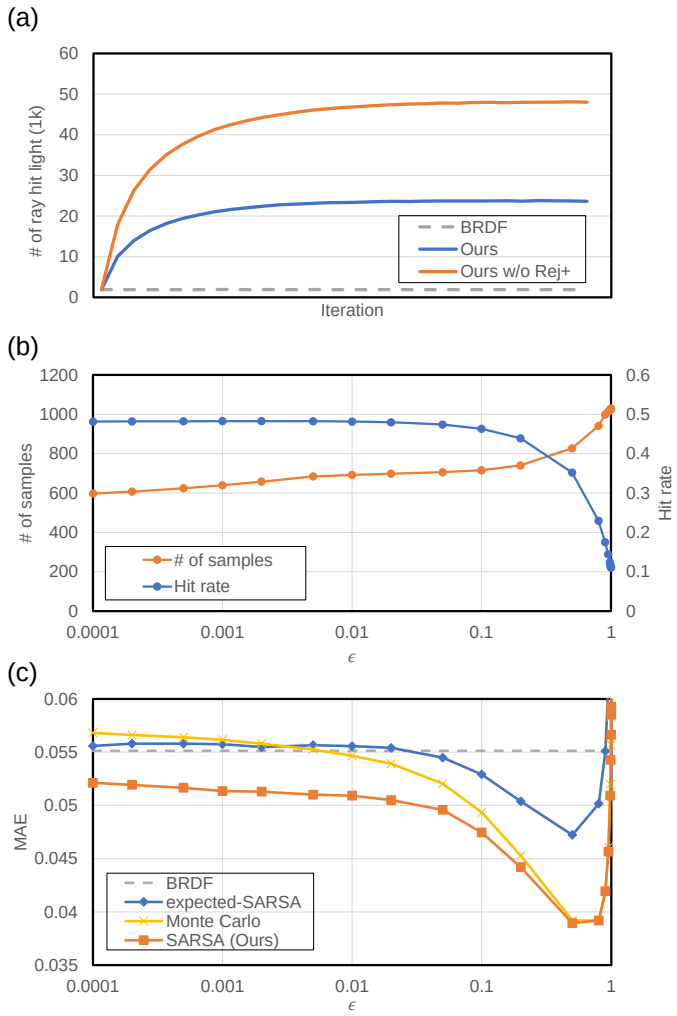

Figure 6.3 Numerical analysis on various aspects of rejection sampling with mixed distribution. (a) The light hit rate increases as number of iteration increases, or the radiance distribution is learned. (b) The error in the rendered image changes as the mixture ratio of two distributions changes for the rejection sampling. SARSA has the minimal error when using the correct ϵ. (c) The trade-off between the hit rate and the number of samples. The hit rate is high with small ϵ while the number of valid samples might decreases.

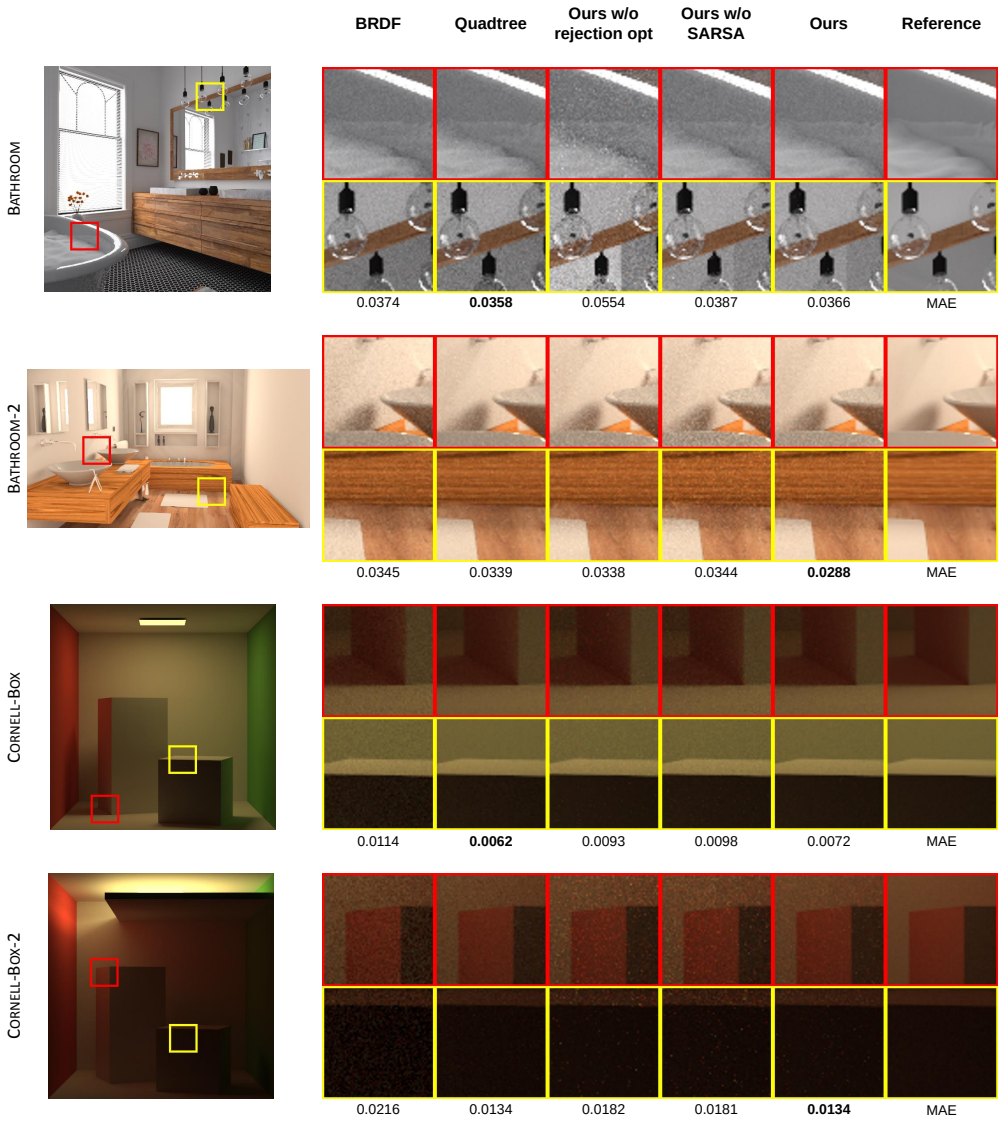

Figure 6.4 Qualitative result for equal time comparison. Each column refers to standard path tracer with BRDF sampling, our proposed method, our proposed method without rejection optimization, our proposed method without SARSA (expected-SARSA) was used instead as [2]) and quadtree based sampling [14] with MC learning.

Figure 6.5 Continue of Figure 6.4.

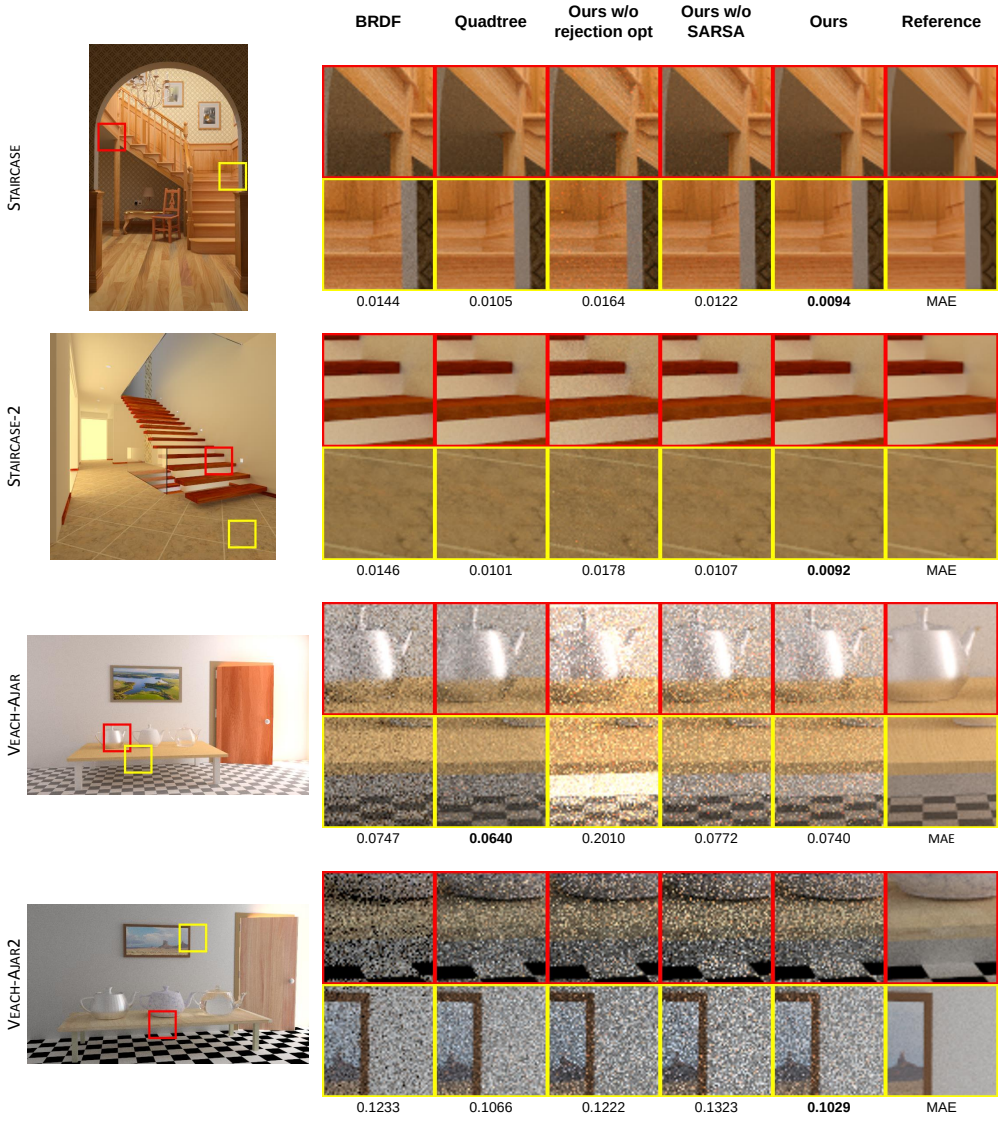

Figure 6.6 Continue of Figure 6.4.

Chapter 7

Conclusion

In this work, we propose a fast and memory-efficient path guiding algorithm in the GPU environment. We divided path guiding into two alternating tasks; learning radiance and sampling radiance. For learning radiance, we suggest a SARSA-based update which outperforms the expected-SARSA or Monte Carlo method. SARSA has a low computational cost since it does not include aggregation over the next actions as expected-SARSA, but also has low memory usage in estimation compared to the Monte Carlo method. For sampling radiance, we only sample in the valid hemisphere from spherical distribution using rejection sampling. Furthermore, we mixed the sampling distribution with randomness to reduce the rejection rate and exploit memoization to avoid repeated computation. All of our suggested methods have been implemented on GPU with megakernel architecture using OptiX [19]. However, our work is designed to also work on wavefront-based rendering which could be covered in future work. Although we used a simple grid-shaped data structure, a more sophisticated data structure for GPU could be investigated.

Appendix A

Additional Experimental Results

In this chapter, we present additional experimental results that are not discussed in the main manuscript.

A.1 Comparison for Spatial Directional Resolution

Table A.1 shows MAE for different spatial, directional resolutions. We only tested our proposed algorithm using SARSA and optimized rejection sampling. For faster comparison, we halve the image size, so the error value is different from the main paper. Spatial resolution 4 and directional resolution 16 gave the best result, we think that 4 is too small, so used 8 instead for the main experiments.

A.2 Equal SPP Comparison

Table A.2 shows MAE for equal spp (1024) budget. For BRDF method and quadtree method (MC), error is 0.0645 and 0.0522 each. Note that unlike the

Table A.1 Comparison for different spatial directional resolutions. S means spatial and D means directional in the table.

S \ D	32	16	8	4
32	0.0936	0.0564	0.0435	0.0466
16	0.0489	0.0380	0.0367	0.0448
8	0.0376	0.0347	0.0362	0.0443
4	0.0351	**0.0344**	0.0363	0.0461

equal-time budget, SARSA does not give the best result. Another thing to note is that Rej+ gives better results than Rej even though Rej shows a higher light hit rate. This seems because of the error in calculating the normalizing term. Since we use stratified Monte Carlo integration to calculate the normalizing term, if sampling radiance distribution is highly unbalanced it causes a higher error, and mixing uniform function helps to reduce the error.

Table A.2 Comparison for equal spp(1024) budget. For BRDF method and quadtree method (MC), error is 0.0645 and 0.0522 each.

MAE	Sphere	Hemisphere		
	Inv	Inv	Rej	Rej+
Expected -SARSA	0.0501	0.0466	0.0531	0.0391
MC	0.0482	0.0495	0.0543	0.0380
SARSA	0.0504	0.0475	0.0551	0.0392

Appendix B

Pseudocode for the Algorithm

We provide pseudocode of our algorithms for the following 5 sampling methods.

- Spherical domain, inversion sampling
- Hemispherical domain, inversion sampling
- Hemispherical domain, rejection sampling
- Hemispherical domain, rejection sampling with optimization
- Hemispherical domain, Metropolis sampling

Each algorithm can be found in Algorithm 1 to 5. Here, η is a uniform random variable in $[0, 1]$, $N(x, \omega, n)$ is a normalizing term, $\int_\Omega L_i(x, \omega_i) f_r(x, \omega, \omega_i)(n \cdot \omega_i) d\omega_i$ and $p_{max}(x, \omega, n)$ is $\max_{\omega_i} L_i(x, \omega_i) f_r(x, \omega, \omega_i)(n \cdot \omega_i)$. $N(x, \omega, n)$ is calculated with stratified Monte-Carlo integration and $p_{max}(x, \omega, n)$ is found with linear search. For diffuse material, these values can be memoized with 5D table since $N(x, \omega, n) = N(x, n)$ and $p_{max}(x, \omega, n) = p_{max}(x, n)$. $M(x, \omega, n)$ in

Metropolis sampling (Algorithm 5) represents previous state corresponding to given conditions x, ω, n. Again, for diffuse material, we can remove ω dependency, so $M(x, \omega, n)$ could be stored similar to $N(x, \omega, n)$. Also, note that for spherical domain sampling, multiple importance sampling (balance heuristics) is used with a probability of 0.5 to sample from pure BRDF without considering radiance as [14] did for quadtree sampling.

Algorithm 1 Inversion sampling on spherical domain

```
1: procedure InversionSampleSphere(x, ω)
2:     r ← η
3:     return BinarySearch(CDF(x), r)
```

Algorithm 2 Inversion sampling on hemispherical domain

```
1: procedure InversionSample(x, n, ω)
2:     r ← η
3:     v ← 0
4:     for k = 1, 2, ..., N do
5:         p ← L_i(x, ω_k) f_r(x, ω, ω_k)(n · ω_k)/N(x, ω, n)
6:         v ← v + p
7:         if r ≤ v then
8:             return ω_k, p
```

Algorithm 3 Rejection sampling

procedure RejectSample(x, n, ω)
    while True **do**
        $\omega_i \leftarrow$ UniformHemisphere(n)
        $p \leftarrow L_i(x, \omega_i) f_r(x, \omega, \omega_i)(n \cdot \omega_i)/N(x, \omega, n)$
        if $\eta < p/p_{max}(x, \omega, n)$ **then**
            break
    return ω_i, p

Algorithm 4 Rejection sampling with speed optimization

procedure RejectSampleOpt(x, n, ω)
    $c \leftarrow 1/p_{max}(x, \omega, n)$
    $\epsilon \leftarrow \max(\frac{1-2c}{2-2c}, 0)$
    $p_{max} \leftarrow (1 - \epsilon)p_{max}(x, \omega, n) + \epsilon u$
    while True **do**
        $\omega_i \leftarrow$ UniformHemisphere(n)
        $p \leftarrow L_i(x, \omega_i) f_r(x, \omega, \omega_i)(n \cdot \omega_i)/N(x, \omega, n)$
        $p \leftarrow (1 - \epsilon)p + \epsilon u$
        if $\eta < p/p_{max}$ **then**
            break
    return ω_i, p

Algorithm 5 Metropolis sampling

procedure MetropolisSample(x, n, ω)
    $m \leftarrow M(x, \omega, n)$
    $m' \leftarrow \text{Mutate}(m)$
    $p \leftarrow L_i(x, m) f_r(x, \omega, m)(n \cdot m)/N(x, m, n)$
    $p' \leftarrow L_i(x, m') f_r(x, \omega, m')(n \cdot m')/N(x, m', n)$
    $a \leftarrow \min(1, \frac{p'}{p})$
    if $\eta < a$ **then**
        $M(x, \omega, n), m \leftarrow m'$
        $p \leftarrow p'$
    return m, p

Bibliography

[1] Benedikt Bitterli. *Rendering resources*. https://benedikt-bitterli.me/resources/. 2016.

[2] Ken Dahm and Alexander Keller. "Learning light transport the reinforced way". In: *ACM SIGGRAPH 2017 Talks*. 2017, pp. 1–2.

[3] Stavros Diolatzis et al. "Practical Product Path Guiding Using Linearly Transformed Cosines". In: *Computer Graphics Forum*. Vol. 39. 4. Wiley Online Library. 2020, pp. 23–33.

[4] Addis Dittebrandt, Johannes Hanika, and Carsten Dachsbacher. "Temporal Sample Reuse for Next Event Estimation and Path Guiding for Real-Time Path Tracing". In: (2020).

[5] Gene Greger et al. "The irradiance volume". In: *IEEE Computer Graphics and Applications* 18.2 (1998), pp. 32–43.

[6] Sebastian Herholz et al. "Product importance sampling for light transport path guiding". In: *Computer Graphics Forum*. Vol. 35. 4. Wiley Online Library. 2016, pp. 67–77.

[7] Heinrich Hey and Werner Purgathofer. “Importance sampling with hemispherical particle footprints”. In: *Proceedings of the 18th spring conference on Computer graphics*. 2002, pp. 107–114.

[8] Henrik Wann Jensen. “Importance driven path tracing using the photon map”. In: *Eurographics Workshop on Rendering Techniques*. Springer. 1995, pp. 326–335.

[9] James T Kajiya. “The rendering equation”. In: *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*. 1986, pp. 143–150.

[10] Alexander Keller and Ken Dahm. *Integral Equations and Machine Learning*. 2019. arXiv: 1712.06115 [cs.LG].

[11] Eric P Lafortune and Yves D Willems. “A 5D tree to reduce the variance of Monte Carlo ray tracing”. In: *Eurographics Workshop on Rendering Techniques*. Springer. 1995, pp. 11–20.

[12] Samuli Laine, Tero Karras, and Timo Aila. “Megakernels considered harmful: Wavefront path tracing on GPUs”. In: *Proceedings of the 5th High-Performance Graphics Conference*. 2013, pp. 137–143.

[13] Thomas Müller. ““Practical Path Guiding” in Production”. In: *ACM SIGGRAPH Courses: Path Guiding in Production, Chapter 10*. Los Angeles, California: ACM, 2019, 18:35–18:48. DOI: 10.1145/3305366.3328091.

[14] Thomas Müller, Markus Gross, and Jan Novák. “Practical path guiding for efficient light-transport simulation”. In: *Computer Graphics Forum*. Vol. 36. 4. Wiley Online Library. 2017, pp. 91–100.

[15] Thomas Müller et al. “Neural control variates”. In: *ACM Transactions on Graphics (TOG)* 39.6 (2020), pp. 1–19.

[16] Thomas Müller et al. “Neural importance sampling”. In: *ACM Transactions on Graphics (TOG)* 38.5 (2019), pp. 1–19.

[17] Merlin Nimier-David et al. “Mitsuba 2: A retargetable forward and inverse renderer”. In: *ACM Transactions on Graphics (TOG)* 38.6 (2019), pp. 1–17.

[18] Merlin Nimier-David et al. “Radiative backpropagation: an adjoint method for lightning-fast differentiable rendering”. In: *ACM Transactions on Graphics (TOG)* 39.4 (2020), pp. 146–1.

[19] Steven G Parker et al. “Optix: a general purpose ray tracing engine”. In: *Acm transactions on graphics (tog)* 29.4 (2010), pp. 1–13.

[20] Alexander Rath et al. “Variance-aware path guiding”. In: *ACM Transactions on Graphics (TOG)* 39.4 (2020), pp. 151–1.

[21] Richard S Sutton, Andrew G Barto, et al. *Introduction to reinforcement learning*. Vol. 135. MIT press Cambridge, 1998.

[22] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[23] Jiří Vorba and Jaroslav Křivánek. “Adjoint-driven Russian roulette and splitting in light transport simulation”. In: *ACM Transactions on Graphics (TOG)* 35.4 (2016), pp. 1–11.

[24] Jiří Vorba et al. “On-line learning of parametric mixture models for light transport simulation”. In: *ACM Transactions on Graphics (TOG)* 33.4 (2014), pp. 1–11.

초록

본 연구는 GPU 상에서 작동하는 간단하지만 효과적인 path guiding 알고리즘을 제안한다. Path guiding은 path tracing의 노이즈를 줄이기 위해 제안된 기법으로 샘플링 과정에서 복사 휘도(radiance)를 배우고 이를 이용해 중요도 샘플링(importance sampling)을 수행한다. 복사 휘도의 복잡한 분포를 배우기 위해 이전의 논문들에서는 복잡한 재귀적 데이터 구조를 제안하고 이를 순차적으로 업데이트 하였지만 이는 CPU상에서의 path tracing만을 가정한 것으로 GPU상에서는 쉽게 구현하기 어려우며 효과적으로 작동하지 않는다. 본 논문에서는 GPU 친화적인 간단한 그리드 형태의 데이터를 사용해 path guiding 알고리즘을 진행하였다. 또한 path guiding의 두 가지 목표-(1) 복사 휘도 학습과 (2) 학습된 복사 휘도 분포를 이용한 중요도 샘플링-를 GPU 상에서 효과적으로 구현하기 위해 다음과 같은 방법을 제시한다. 우선 복사 휘도 학습의 경우, 강화학습과 복사 휘도 학습의 구조적 유사성을 밝힌 이전 연구 [2]를 확장하여 가볍고 빠른 SARSA [22]를 이용한 학습 방법을 제안하였다. 학습된 복사 휘도는 공간-방향을 그리드 형태로 분할한 GPU상의 데이터 구조에 저장된다. 학습된 복사 휘도를 사용한 중요도 샘플링의 경우 법선 벡터 방향에 유효하지 않은 샘플들은 제외한 뒤, 리젝션 샘플링(rejection sampling)을 이용해 중요도 샘플링(importance sampling)을 수행하였다. 모든 알고리즘은 NVIDIA OptiX [19]를 사용해 GPU상에서 megakernel 구조로 구현되었다. 복잡한 구조의 씬 데이터에 대해 여러번 실험을 수행하였으며 본 연구에서 제안한 방법의 우수성을 확인하였다.

주요어: Path Guiding, 강화학습, 광선 추적법

학번: 2019-27633

Acknowledgements

연구 및 논문 작성 과정에서 많은 도움을 주신 김영민 교수님께 감사드린다. 또한 3D vision 연구실 학생들에게도 감사를 표하고 싶다.