Pacific Graphics (2021)
M. Okabe, S. Lee, B. Wuensche and S. Zollmann (Editors)

Short Paper

Fast and Lightweight Path Guiding Algorithm on GPU

Juhyeon Kim[1] and Young Min Kim[1]

[1]Department of Electrical and Computer Engineering, Seoul National University

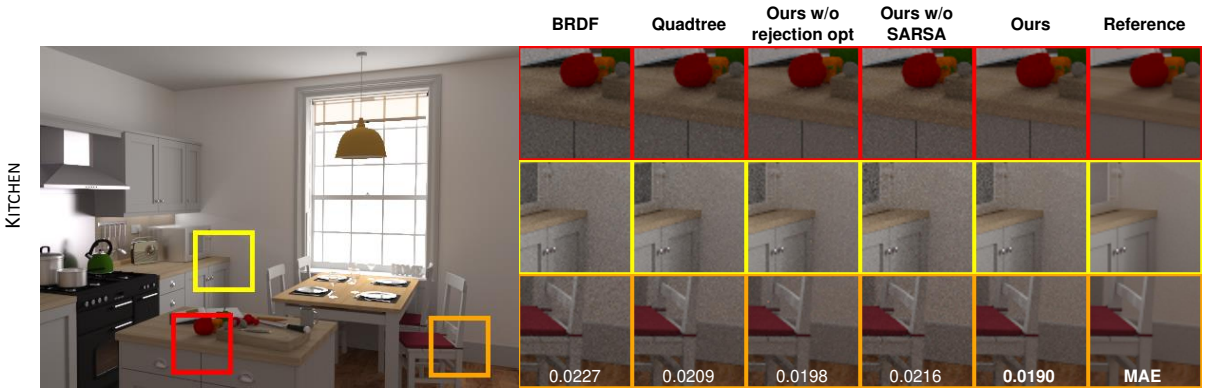

Figure 1: *We present a fast and lightweight path guiding algorithm implemented in GPU. Our algorithm utilizes a regular grid structure and combines the RL algorithm to learn the radiance distribution. The learned distribution is then combined for the product importance sampling of path guiding, from which we can produce a photo-realistic image. When we compare the path-traced images produced in an equal amount of time, our algorithm produces superior results. The mean absolute error (MAE) values are included in the last row.*

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 [SB18] 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 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 [PBD*10]. 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.*

CCS Concepts

• ***Computing methodologies*** → ***Ray tracing;*** *Reinforcement learning; Massively parallel algorithms;*

1. Introduction and Background

Path tracing is a Monte-Carlo method that faithfully simulates light transport to synthesize a photo-realistic image. The rendering equation [Kaj86] describes the outgoing radiance $L_o(x,\omega)$ from point x in direction ω as

$$L_o(x,\omega) = L_e(x,\omega) + \int_\Omega L_i(x,\omega_i) f_r(x,\omega,\omega_i)(n\cdot\omega_i)\mathrm{d}\omega_i. \quad (1)$$

The first term $L_e(x,\omega)$ is the emitting radiance and the second term aggregates the reflection of incoming radiance over all the incoming directions ω_i over hemisphere Ω. The proportion reflected into the outgoing direction ω is defined with the bidirectional reflectance distribution function (BRDF) $f_r(x,\omega,\omega_i)$ and the inner product between the surface normal n and the incoming direction ω_i. Path tracing evaluates the above rendering equation using

DOI: 10.2312/pg.20211379

Monte Carlo integration with N samples

$$\langle L_o(x,\omega)\rangle = L_e(x,\omega) + \frac{1}{N}\sum_{j=1}^{N}\frac{L_i(x,\omega_j)f_r(x,\omega,\omega_j)(n\cdot\omega_j)}{p(\omega_j|x,\omega)}, \quad (2)$$

where $p(\omega_j|x,\omega)$ is the sampling PDF. *Path guiding* accelerates path tracing by iteratively (i) learning high-energy light paths and (ii) sampling according to the learned distribution. Although a considerable amount of research has been conducted on path guiding [Jen95; HP02; VKŠ*14; MGN17], most of them are implemented on the CPU and the GPU case has been rarely studied except a few cases [DHD20]. With the emergence of GPU ray tracing libraries such as NVIDIA OptiX [PBD*10], more commercial ray tracing programs will shift to GPU-based versions to take advantage of massively parallel processing.

In this paper, we propose a practical path guiding algorithm that can be incorporated into the existing GPU path tracing pipeline. We adapt the framework to be parallelizable by employing regular data structure and concurrent updates of the distribution. As illustrated in Figure 2, the scene is divided into voxels, and the directions are discretized using the equal-area projection with regular shape as proposed in [GSHG98]. We learn the radiance field with a fast and lightweight reinforcement learning (RL) algorithm called SARSA as described in Section 2. Also, we suggest a rejection sampling method to quickly advocate for the fine geometry and overcome the limitation of coarse grid resolution as presented in Section 3. We demonstrate that the proposed method can accelerate the path-guiding algorithm in various scenes.

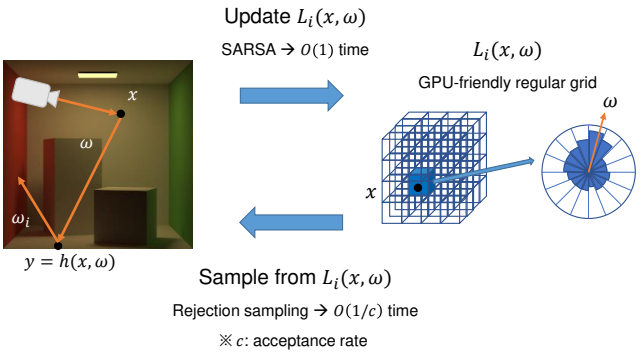

Figure 2: *The overall flow of our proposed method. We store incident radiance $L_i(x,\omega)$ in GPU-friendly regular data structure. We process rays and efficiently update $L_i(x,\omega)$ with SARSA, while, from $L_i(x,\omega)$, we quickly sample valid rays to render the scene with rejection sampling.*

2. Fast and Lightweight Radiance Learning

The rendering equation in Equation 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 of the ray originated from x to direction ω. Using this representation, we can rewrite the rendering equation in a recurrent form using the incident radiance L_i:

$$L_i(x,\omega) = L_e(y,-\omega) + \int_\Omega L_i(y,\omega_i)f_r(y,-\omega,\omega_i)(n\cdot\omega_i)\mathrm{d}\omega_i. \quad (3)$$

While coming from a different context, the recursive equation in Equation 3 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:

$$\begin{aligned} Q(s,a) \leftarrow{}& (1-\alpha)\cdot Q(s,a) \\ &+\alpha\cdot\left(r(s,a,s') + \gamma\int_A \pi(s',a')Q(s',a')\mathrm{d}a'\right), \end{aligned} \quad (4)$$

where $\pi(s',a')$ (also known as the *policy*) is a probability to choose next action a' at the next state s' and α is a learning rate. Because of structural similarity between Equation 3 and the update target in Equation 4, learning radiance distribution $L_i(x,\omega)$ can be achieved using expected-SARSA with $\gamma = 1$ [DK17]. Regarding the integral term, [DK17] 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 3) can be updated as following

$$\begin{aligned} &L_e(x^{(m+1)},-\omega^{(m)}) \\ &\quad+\frac{2\pi}{N}\sum_{k=1}^{N} L_i(x^{(m+1)},\omega_k)f_r(x^{(m+1)},-\omega^{(m)},\omega_k)(n\cdot\omega_k), \end{aligned} \quad (5)$$

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

Instead, taking inspiration from the frameworks in RL, we propose a fast and lightweight method to learn the complex distribution of radiance. Basically, in RL, Q function estimation could be largely categorized into three groups; dynamic programming (DP), Monte Carlo (MC), and temporal difference (TD) methods. Figure 3 illustrates how the aforementioned Q value prediction algorithms can be interpreted in radiance learning. DP, or expected-SARSA, involves the exhaustive aggregation over the next state which is computationally heavy (Figure 3(a)). MC method updates Q value to the actual return from complete episodes without bootstrapping (Figure 3(b)). However, MC requires an additional memory because we have to store all previous points which can be a serious problem when we concurrently process a large number of rays. Instead, we propose to use SARSA, one of TD methods that updates Q value to the expected future return with bootstrapping.

Unlike expected-SARSA, SARSA only considers the single next state, which does not require aggregation (Figure 3(c)). Specifically, the SARSA's update target is similar to expected-SARSA, while it only differs in estimating future expected rewards

$$L_e(x^{(m+1)},-\omega^{(m)}) + a^{(m+1)}L_i(x^{(m+1)},\omega^{(m+1)}), \quad (6)$$

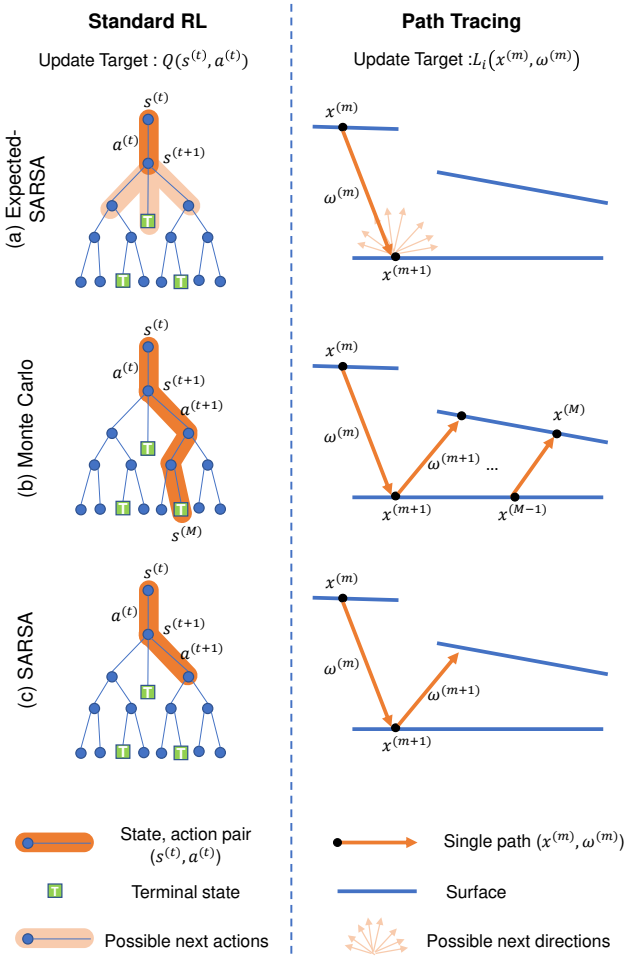

Figure 3: *Difference between three updating method (a) expected-SARSA, (b) Monte Carlo and (c) SARSA in standard RL and path tracing.*

where $a^{(j)} = \frac{f_r(x^{(j)},\omega^{(j-1)},\omega^{(j)})(n\cdot\omega^{(j)})}{p(\omega^{(j)}|x^{(j)},\omega^{(j-1)})}$ is the attenuation factor. By considering only a single next state, the time complexity could be reduced from $O(N)$ to $O(1)$ where N is a number of the possible states. 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, SARSA is superior to expected-SARSA or MC in terms of speed and memory consumption, which can be exploited with GPU acceleration.

The update in Equation 6 is attenuated by α, yielding the full update equation

$$L_i(x,\omega) \leftarrow (1-\alpha)\cdot L_i(x,\omega) + \alpha\left(L_e(y,-\omega) + a_y L_i(y,\omega_i)\right) \quad (7)$$

where a_y is the attenuation factor at $(y,-\omega,\omega_i)$. However such update still suffers from the concurrency issue, and possibly yield 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.

3. Efficient Importance Sampling from Learned Radiance

As we update the radiance value, we simultaneously run path guiding in Equation 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 discretized angle. Note that our sampling distribution jointly considers the radiance, BRDF, and the cosine term such that we can effectively perform product importance sampling. The 5D radiance field $L_i(x,\omega)$ is tabulated as a 3D coarse spatial grid that contains the 2D spherical radiance distribution in each cell as shown in Figure 4. Using the shared spherical distribution per spatial cell reduces memory but results in invalid samples that are directed opposite to the surface normal. Previous works [DK17; VKŠ*14] store hemispherical distributions adaptive to each surface point that align with the surface normal to achieve better sampling efficiency at the cost of additional memory.

We propose a hybrid approach to efficiently sample the distribution considering the local geometry, which greatly reduces the number of wasted samples. 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 4. While our GPU-friendly grid structure may suffer from lower directional resolution than the quadtree-based implementation [MGN17], 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. The distribution can be sampled in many ways, but let's only think of three widely used ways.

Inversion sampling The most intuitive way is *inversion method*, which samples from the cumulative density function (CDF). Since we are sampling from the intersection of the normal-oriented hemisphere and the spherical distribution, and we have to dynamically construct the CDF 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. In our setting, we use the uniform distribution $u(\omega)$ whose domain is the normal-oriented hemisphere. The sample is accepted 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 minimum value $(1/p_{\max})$, then c be-

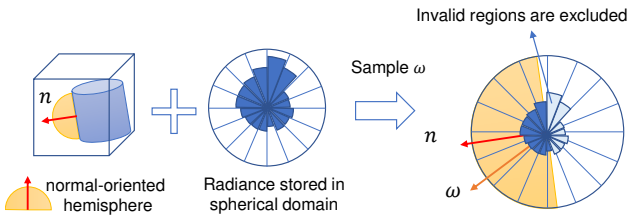

Figure 4: *The hemispherical domain sampling removes probability of sampling from invalid hemisphere.*

comes the same as the acceptance rate which represents the relative area between the two distributions (Figure 5, left).

In order to reduce rejection rate, we propose to mix a sampling distribution with uniform distribution:

$$p_{\text{sampling}} = (1-\varepsilon)p + \varepsilon u, \tag{8}$$

where $\varepsilon \in [0,1]$ is a constant value that controls the trade-off between using the correct distribution for the importance sampling and the high rejection rate. Figure 5 illustrates how the mixture maintains the shape of the original distribution but yet avoids severe rejection by smoothing the peaks. Numerically the total number of effective samples is

$$\underset{\varepsilon\in[0,1]}{\arg\max}(1-\varepsilon)\,(c+(1-c)\varepsilon) \tag{9}$$

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 4.3.

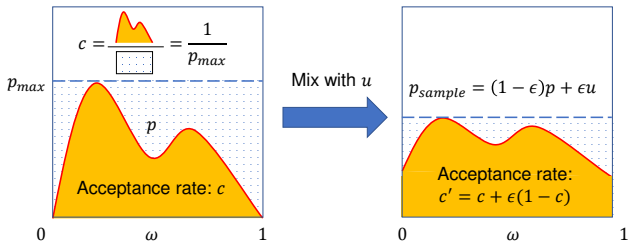

Figure 5: *Rejection rate alleviation with mixing uniform PDF.*

Metropolis sampling *Metropolis sampling* is a Monte Carlo Markov Chain algorithm, where samples are drawn from an arbitrary mutation function and then the samples are mutated with a pre-defined probability. However, such sequential mutation on a random variable cannot be easily implemented on GPU, possibly causing race conditions that update the same random variable simultaneously. Thus, it is skipped in this paper.

We can therefore perform sampling with adaptive normal directions on GPU via rejection sampling. 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 sample 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 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 applied 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 $L_i(x,\omega)$.

Table 1: *Equal time comparison for several methods. The top part shows the mean absolute error. BRDF-based method samples the ray according to BRDF and does not consider the radiance distribution, and quadtree-based method is our implementation of [MGN17] on GPU. 'Ours without Rej+' samples the distribution without rejection optimization. 'Ours without SARSA' utilizes expected-SARSA for radiance learning [DK17].*

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

4. Experiments and Results

We implemented our algorithm on a GPU environment with megakernel architecture. We wrote the path-guiding algorithm with our own renderer using OptiX [PBD*10] and built several BRDFs referring to the rich material library of Mitsuba2 [NVZJ19].

We tested our algorithm for 12 scenes from [Bit16]. All of the path guiding methods used unidirectional path tracing without *next event estimation* (NEE) for simplicity as [MGN17] and the following works. 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 or *samples per pixel* (spp) budget can be imposed, but a time budget (40 sec) 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 [MGN17], 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. 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.

4.1. 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 [MGN17] in Table 1. 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 0.5, the same with the original paper. The maximum number of nodes is set to the same as the grid case. Also, as the original paper, it is set to use MC method only to learn radiance.

BRDF-based method can be easily implemented on GPU and fast, leading to process more number of samples for equal-time comparison. However, 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 result than pure BRDF sampling, but seems to suffer from invalid samples that heading down to the surface. On the other hand, our implementation of path guiding algorithm is normal-sensitive, thus provides nearly zero invalid samples. It has 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 7-(a). Compared to BRDF sampling, our method achieves $10 \sim 20$ times higher hit rate.

4.2. Comparison for Radiance Learning Methods (Section 2)

In this section, we compare several radiance learning method discussed in Section 2, which are namely based on expected-SARSA, Monte Carlo and SARSA. Table 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 due to accessing a record that stores previous points.

Figure 6 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 [SB18]. We can easily verify this in Figure 6 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 minimal affect in the final image, and speed is more important factor when time becomes the budget.

Memory consumption is also an important issues 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 [MGN17]) which may requires 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 total 12 floats for single data, 16 maximum depth, and about 46,000 concurrent rays, indicating that Monte-Carlo method causes additional 35 MB of stack usage. Of course this may still harmful for performance, the more serious problem occurs when we use 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.

Table 2: *Equal time (40 sec) comparison for different learning and sampling method discussed in Section 2 and Section 3.*

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

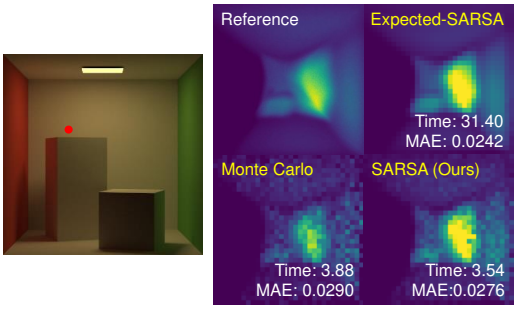

Figure 6: *The learned radiance map at the position indicated as a red dot in the scene on the left. MAE and required time per sample (ms) are showed. We increased directional grid resolution to emphasize the difference.*

4.3. Comparison for Radiance Sampling Methods (Section 3)

In this section, we compare the radiance sampling methods covered in Section 3. The simplest way is the inversion method using the stored spherical distribution without considering normal which is similar to quadtree method. 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.

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. 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

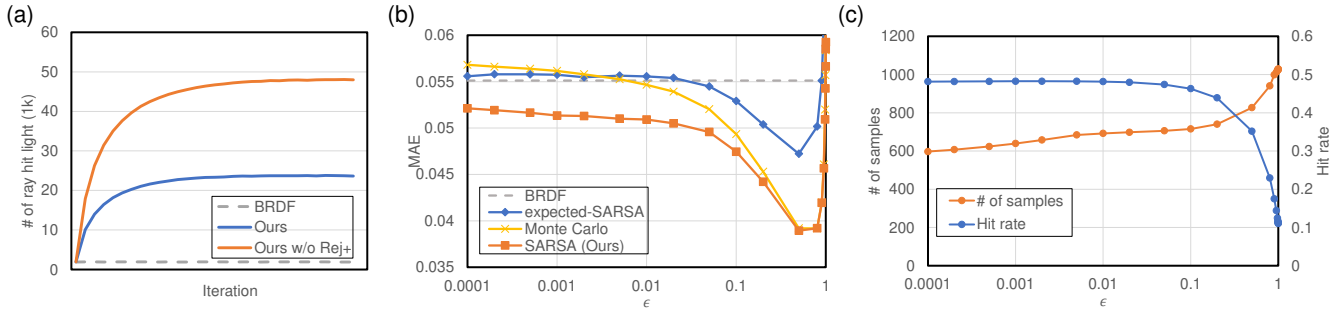

Figure 7: *Numerical analysis on various aspects of rejection sampling with mixed distribution. (a) The light hit rate increases as number of iteration increases, or the radiance distribution is learned. (b) The error in the rendered image changes as the mixture ratio of two distributions changes for 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.*

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 9. With the optimized ε as shown in Equation 9 (indicated with post-fixed '+' sign in Table 1 and 2), the rejection sampling 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 $c' = c + (1-c)\varepsilon$ is an acceptance rate for the mixed PDF as proposed in Section 3.

Effect of ε in Equation 8 We further investigate the effect of mixing the sampling distributions with different $\varepsilon \in [0.0001, 1]$ in Figure 7. Figure 7-(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 7-(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 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 7-(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.

5. Conclusion

We propose a fast and memory-efficient path guiding algorithm in the GPU environment. For learning radiance, we suggest SARSA-based update which outperforms expected-SARSA or Monte Carlo method. For sampling radiance, we only sample in the valid hemisphere from spherical distribution employing rejection sampling and memoization to achieve efficient and fast sampling. All of our suggested methods have been implemented on GPU with megakernel architecture using OptiX [PBD*10]. However, our work is designed to also work on wavefront-based rendering which could be covered in future work.

References

[Bit16] BITTERLI, BENEDIKT. *Rendering resources*. https://benedikt-bitterli.me/resources/. 2016 4.

[DHD20] DITTEBRANDT, ADDIS, HANIKA, JOHANNES, and DACHSBACHER, CARSTEN. "Temporal Sample Reuse for Next Event Estimation and Path Guiding for Real-Time Path Tracing". (2020) 2.

[DK17] DAHM, KEN and KELLER, ALEXANDER. "Learning light transport the reinforced way". *ACM SIGGRAPH 2017 Talks*. 2017, 1–2 2–4.

[GSHG98] GREGER, GENE, SHIRLEY, PETER, HUBBARD, PHILIP M, and GREENBERG, DONALD P. "The irradiance volume". *IEEE Computer Graphics and Applications* 18.2 (1998), 32–43 2.

[HP02] HEY, HEINRICH and PURGATHOFER, WERNER. "Importance sampling with hemispherical particle footprints". *Proceedings of the 18th spring conference on Computer graphics*. 2002, 107–114 2.

[Jen95] JENSEN, HENRIK WANN. "Importance driven path tracing using the photon map". *Eurographics Workshop on Rendering Techniques*. Springer. 1995, 326–335 2.

[Kaj86] KAJIYA, JAMES T. "The rendering equation". *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*. 1986, 143–150 1.

[MGN17] MÜLLER, THOMAS, GROSS, MARKUS, and NOVÁK, JAN. "Practical path guiding for efficient light-transport simulation". *Computer Graphics Forum*. Vol. 36. 4. Wiley Online Library. 2017, 91–100 2–5.

[NVZJ19] NIMIER-DAVID, MERLIN, VICINI, DELIO, ZELTNER, TIZIAN, and JAKOB, WENZEL. "Mitsuba 2: A retargetable forward and inverse renderer". *ACM Transactions on Graphics (TOG)* 38.6 (2019), 1–17 4.

[PBD*10] PARKER, STEVEN G, BIGLER, JAMES, DIETRICH, ANDREAS, et al. "Optix: a general purpose ray tracing engine". *Acm transactions on graphics (tog)* 29.4 (2010), 1–13 1, 2, 4, 6.

[SB18] SUTTON, RICHARD S and BARTO, ANDREW G. *Reinforcement learning: An introduction*. MIT press, 2018 1, 5.

[VKŠ*14] VORBA, JIŘÍ, KARLÍK, ONDŘEJ, ŠIK, MARTIN, et al. "On-line learning of parametric mixture models for light transport simulation". *ACM Transactions on Graphics (TOG)* 33.4 (2014), 1–11 2, 3.