Advanced (Quasi)-Monte Carlo Methods for Image Synthesis

Siggraph 2012 Course

Image courtesy Delta Tracing with NVIDIA iray.

PRESENTERS

Leonhard Grünschloß
Weta Digital

Alexander Keller
NVIDIA ARC GmbH

Simon Premože
Double Negative

Matthias Raab
NVIDIA ARC GmbH

COURSE DESCRIPTION

Monte Carlo ray tracing has become ubiquitous in most commercial renderers and in custom shaders used for visual effects and feature animation. However, many advanced Monte Carlo algorithms are not widely used and are often misunderstood. Course attendees will learn about the practical aspects of variance reduction methods with a focus on all variants of importance sampling. The course also covers quasi-Monte Carlo methods at industry level, as well as the practical aspects of bidirectional path tracing combined with multiple importance sampling and Metropolis Light Transport. The audience will benefit from the intuition provided by the practical advice throughout the course.

Level of difficulty: advanced

COURSE LENGTH

Half-day course.

INTENDED AUDIENCE

Renderer engineers, shader writers, and students with an interest in physically-based rendering, efficient simulation technology, and practical advice.

PREREQUISITES

Basic understanding of raytracing and light transport. Understanding of probability and linear algebra.

Leonhard Grünschloß is working on physically based rendering at Weta Digital. He received his Master's degree in Computer Science at Ulm University in 2008. Afterwards he worked at *mental images* for three years, where he was designing new rendering architectures and shading languages. His main research interests include efficient and robust sampling techniques for photorealistic image synthesis in the context of large production scenes, with a focus on quasi-Monte Carlo methods.

Alexander Keller is a member of NVIDIA Research and leads advanced rendering research at NVIDIA ARC GmbH, Berlin. Before, he had been the Chief Scientist of *mental images* and had been responsible for research and the conception of future products and strategies including the design of the *iray* renderer. Prior to industry, he worked as a full professor for computer graphics and scientific computing at Ulm University, where he co-founded the UZWR (Ulmer Zentrum für Wissenschaftliches Rechnen). He holds a PhD in computer science and he authored more than 21 patents, and published more than 40 papers mainly in the area of quasi-Monte Carlo methods and photorealistic image synthesis.

Simon Premože is currently writing physically-inspired shaders at Double Negative. He graduated from the University of Utah where he primarily studied appearance models, volume rendering and global illumination. Previously, he was an R&D engineer at Industrial Light and Magic where he worked on a variety of rendering problems in production. His current research interests include interactive global illumination and rendering algorithms, Monte Carlo methods, modeling natural phenomena and reflectance models.

Matthias Raab is working at NVIDIA's Advanced Rendering Center in Berlin, where he is one of the key engineers of NVIDIA *iray*. Before moving to industry in 2007 and working on the mental ray renderer, he had been a researcher at Ulm university, where he received his Master's degree in Computer Graphics in 2005. Matthias Raab has a strong background in mathematics and scientific computing, especially in Monte Carlo methods. His current work is focussing on variance reduction methods and light transport simulation algorithms.

COURSE SYLLABUS

1. Introduction [Premože/Keller]

a. Monte Carlo and Quasi-Monte Carlo methods [Keller]

b. Applications to Light Transport [Premože]

2. Variance reduction methods [Premože]

a. Overview of Variance Reduction Techniques

b. Importance Sampling

Principles
What works and what doesn't work and why
Weighted Importance Sampling
Multiple Importance Sampling
Deterministic Mixture Sampling
Adaptive Multiple Importance Sampling

c. Control Variates

Practical applications
Approximating visibility

d. Other Techniques

Separation of the main part
Correlated Sampling
Adaptive Sampling

3. QMC Methods in Photorealistic Image Synthesis [Keller / Grünschloß]

a. Consistent vs. Biased vs. Unbiased

b. Quasi-Monte Carlo Points

Halton sequence and Hammersley points
(t,s)-sequences and (t,m,s)-nets in base b
Rank-1 lattice sequences and rank-1 lattices

c. Quasi-Monte Carlo Rendering Techniques

Hybrid sequences
Path tracing
Anti-aliasing
Motion blur
Bidirectional scattering distribution functions
Connecting path segments by shadow rays
Connecting path segments by proximity

4. Bidirectional Path Tracing (BDPT) [Premože]

Connecting path tracing and light tracing
Conversion of densities for Multiple Importance Sampling
Vertex merging
Implementation details
Issues with Bidirectional Path Tracing

5. Metropolis Light Transport (MLT) [Raab]

Metropolis Sampling Algorithm
Application to Light Transport
Strengths & Weaknesses
(Implementing) Mutation Strategies

6. Conclusion and Questions [All]

Advanced (Quasi-) Monte Carlo Methods for Image Synthesis

Course Notes

Leonhard Grünschloß
Alexander Keller
Simon Premože
Matthias Raab

May 2012

Abstract

Monte Carlo ray tracing has become ubiquitous in most commercial renderers and in custom shaders used for visual effects and feature animation. However, many advanced Monte Carlo algorithms are not widely used and are often misunderstood. Course attendees learn about variance reduction methods ranging from importance sampling and its derivatives to control variates and correlated sampling. Audience also learns about Quasi Monte Carlo methods, deterministic pseudo-random number generation and how to correctly incorporate them into a raytracer. Last part of the course is devoted to advanced algorithms such as bidirectional path tracing and Metropolis Light Transport with special emphasis on how to implement these algorithms in practice.

Contents

1 Introduction to Monte Carlo Methods

1.1 Monte Carlo Methods

The term *Monte Carlo* refers to all methods that use a statistical sampling processes to approximate solutions to quantitative problems. It can be used for a wide variety of probabilistic problems ranging from numerical integration to optimization. These methods are used in many application domains such as economics, robotics and nuclear engineering.

In this section, we describe some basic concepts of Monte Carlo integration. After a brief overview of the Monte Carlo methods, we introduce the principle of Monte Carlo integration and look at some of the basic statistical properties. Then, we describe some basic variance reduction techniques, such as importance sampling, control variates, and mixture sampling, that we use in later sections in the context of light transport. This section is only a brief summary, but it does provide some insights and intuition about why some methods perform better than other and describes the circumstances one should use a particular method. We encourage interested readers to learn more about Monte Carlo methods and probability in many excellent books [10, 16, 7] and papers that exist on the topic.

Readers who are mostly interested in the practical implementation Monte Carlo methods for rendering can skip this section.

1.1.1 Estimators

A *continuous random variable* X is a quantity that randomly takes on a value x that lies on the real line $(-\infty, \infty)$. The values of x can be quantitatively described by the *probability density function* (PDF) p. The probability that x will take a value on some interval between a and b is then

$$Pr\{a \leq X \leq b\} = \int_a^b p(x)dx. \tag{1.1}$$

The probability density function $p(x)$ must satisfy two conditions:

1. It is always positive:
$$p(x) \geq 0$$
2. It is normalized:
$$\int_{-\infty}^{\infty} p(x)dx = 1$$

It is important to understand the difference between ***probability*** and the probability density function. The probability, or likelihood, of an event takes values strictly between 0 (***impossible event***, it never happens) and 1 (***certain event***, it always occurs). On the other hand, the probability density function describes the relative likelihood of a random variable (or event) having a certain value. For instance, if $p(x_1) = 10$ and $p(x_2) = 100$, then the random variable with the PDF p is ten times more likely to have a value near x_1 than near x_2. The relationship between the probability density function p and the probability Pr is defined in Equation (1.1).

Other important concepts to understand are expected value and variance of a random variable. The *expectation* (or *expected value*) of a random variable $Y = f(X)$ is

$$E[Y] = \int_\Omega f(x)p(x)dx \tag{1.2}$$

and its variance is

$$V[Y] = E[(Y - E[Y])^2]. \tag{1.3}$$

Intuitively, expected value (or ***mean value***) is just the average value of the random variable. Note that expected value should not be confused with the most probable value. On the other hand, variance measures how much the values of some random variable deviate from its mean or expected value. The higher the variance, the more values differ from the average value.

The expected value has a few useful properties:

1. The expected value of the sum of two random variables X and Y is the sum of the expected values of those variables:

   $$E[X + Y] = E[X] + E[Y]$$

   Since functions of random variables are also random variables, this principle applies to the sum of functions of random variables:

   $$E[f(X) + g(Y)] = E[f(X)] + E[g(Y)]$$

   The above holds true even if variables X and Y are correlated.

2. For any constant a, the expected value and variance for aX are

   $$\begin{aligned} E[aY] &= aE[Y] \\ V[aY] &= a^2V[Y] \end{aligned}$$

If we want to compute an approximation to some unknown quantity Q (i.e.the *estimand* or quantity of interest), a function F of random variables $X_1, \ldots, X_N$ is called an *estimator* if its mean (expected value) $E[F]$ is a usable approximation to Q:

$$F_N = F_N(X_1, \ldots, X_N) \tag{1.4}$$

A particular numerical value of F_N is called an *estimate*. Q can be any function that we might be interested in. In rendering, Q can be the amount of light that reaches a point on a surface or the amount of light reflected from the surface.

There are many possible estimators. In general, we want Monte Carlo estimators to provide good estimates as fast as possible. **How do we choose a good estimator?** First, we need to establish some criteria for what *good* means by looking at the properties of Monte Carlo estimators:

- **Error**

  $$\text{error} = F_N - Q$$

  Mean Square Error (MSE) of an estimator F is then

  $$\text{MSE} = E[(F_N - Q)^2] \tag{1.5}$$

- **Bias**

  Bias β is the expected value of the error:

  $$\beta[F_N] = E[F_N - Q] \tag{1.6}$$

  The estimator is ***unbiased*** if $\beta[F_N] = 0$ for sample size N:

  $$E[F_N] = Q \qquad \text{for all} N \geq 1. \tag{1.7}$$

  An obvious advantage of the unbiased estimator is that we are guaranteed to get the correct value of quantity of interest Q if enough samples are taken. Furthermore, the expected value of an unbiased estimator will be the correct value after any number of samples. The mean square error of the estimator can be also written as

  $$\text{MSE}[F_N] = V[F_N] + \beta[F_N]^2. \tag{1.8}$$

  For unbiased estimators, the MSE is the same as the variance. For biased estimators, the error is much more difficult to estimate. It is also important to know that a biased estimator may not give a correct estimate for Q even if an infinite number of samples are taken. In practice, a biased estimator may have some desirable properties, such as lower variance, which makes it very appealing for use in computer graphics. For example, in rendering, noise is a manifestation of variance. While taking more samples reduces the amount of noise, rendering using a biased estimator may have less noise for the same number of samples albeit producing different images.

- **Consistency**

  An estimator is *consistent* if the error goes to zero as the number of samples grows:

  $$Pr\{\lim_{N \to \infty} F_N = Q\} = 1. \tag{1.9}$$

The above equation is essentially saying that if we use a consistent estimator, we are one hundred percent certain that the answer is correct if we increase the number of samples. Consistency is a stronger condition than requiring the estimator to be unbiased. It is still possible that an unbiased estimator is not consistent, in which case its variance is infinite. A biased estimator is consistent if its bias β decreases to 0 as the number of samples N increases.

1.1.2 Monte Carlo Integration

The basic idea behind Monte Carlo integration is evaluation of the integral

$$I = \int_\Omega f(x)dx \tag{1.10}$$

using random sampling. Here, N random points $X_1, X_2, \ldots, X_N$ are independently sampled from some density function p and used to approximate I,

$$\hat{I}_N = \frac{1}{N}\sum_{i=1}^{N} f(X_i). \tag{1.11}$$

Notation note: A realization of an estimator F, namely F_N, is the same as $\hat{I}_N$. The subscript N emphasizes that $\hat{I}$ is still a random variable and therefore its properties depend on how many samples were chosen.

As N increases, the expected error of this estimate decreases. We want to choose N such that we have confidence that the estimate $\hat{I}_N$ is good. The estimator $\hat{I}_N$ is a crude but unbiased estimator for I and its variance is

$$V[\hat{I}_N] = V[\frac{1}{N}\sum_{i=1}^{N} f(X_i)] = \frac{1}{N}V[f(X)]. \tag{1.12}$$

From this variance estimate $V[\hat{I}_N]$ we can conclude the following:

1. The standard error of the estimator decreases with the square root of the sample size N. Recall that the standard error of $\hat{I}_N$ is $V[\hat{I}_N]^2$, so while the variance of the estimate is proportional to $1/N$ the standard deviation is proportional to $1/\sqrt{N}$. Therefore, to reduce the error in half, we have to quadruple the number of samples.

2. The statistical error is independent of the dimensionality of the integral. This simply means that the computation does not increase exponentially when the dimensionality of the integral increases.

So far, we have not made any assumptions about function $f(x)$ we are trying to integrate. On the other hand, in the above discussion we have assumed that our random variable X is uniformly distributed over the integration domain Ω. Loosely speaking, a uniform distribution implies that the probability of choosing each sample is equal. Unfortunately, real problems are rarely this simple. For example, function $f(x)$ can be zero in many regions and have very high values in other. If uniformly sampling the domain Ω, we may get very large variance. Also, sometimes it may not be possible to sample a space uniformly. In order to alleviate these problems, we can rewrite the estimator from Equation (1.11) as

$$\begin{aligned} I &= \int_\Omega f(x)dx \\ &= \int_\Omega \frac{f(x)}{p(x)} p(x)dx, \end{aligned}$$

where $p(x)$ is a probability density function in Ω. We can now generate N samples from distribution $p(x)$ (instead of uniformly sampling Ω) to get

$$\hat{I}_p = \frac{1}{N}\sum_{i=1}^{N} \frac{f(X_i)}{p(X_i)}. \tag{1.13}$$

The simple Monte Carlo estimator was saw in Equation (1.11) is just a special case of the more general estimator in Equation (1.13) with $p(x)$ being a uniform distribution in Ω. This estimator has the same variance properties we have seen above.

One major advantage of Monte Carlo integration is that it is easy to understand and simple to use. If we can generate random samples using some density $p(x)$ and have the ability to compute the sample weights, $w_i = \frac{f(X_i)}{p(X_i)}, i = 1, \ldots, N$, then we can evaluate the integral. Monte Carlo methods are also flexible, robust and work well in higher dimensions where other numerical methods might fail.

1.1.3 Variance Reduction Techniques

One of the biggest disadvantages of Monte Carlo methods is a relatively slow convergence rate. As we have already discussed above, the root mean square (RMS) error converges slowly at a rate of $O(1/\sqrt{N})$, so we need to quadruple number of samples N to halve the error.

Ideally, we would like to use an estimator which has both small variance and is computationally efficient. *Efficiency* of a Monte Carlo estimator F is

$$\epsilon[F] = \frac{1}{V[F]T[F]} \tag{1.14}$$

where $V[F]$ is the variance and $T[F]$ is the time needed to evaluate F. Therefore, the more efficient the estimator is the lower the variance in a given (fixed) amount of time.

One of the fundamental goals in researching Monte Carlo methods is to find or design efficient estimators. These techniques are often called *variance reduction techniques* and include *importance sampling*, *control variates*, and *adaptive sampling*. We briefly review some of these techniques. In later sections, we apply these techniques to the direct illumination rendering problem.

Importance Sampling

Recall that a Monte Carlo estimator for some function $f(x)$ over domain Ω is

$$I = \int_\Omega \frac{f(x)}{p(x)} p(x) dx$$

and the estimator is

$$\hat{I}_p = \frac{1}{N} \sum_{i=1}^{N} \frac{f(X_i)}{p(X_i)}.$$

The variance of the estimator $\hat{I}_p$ depends on the density $p(x)$ from which random samples are drawn. If we choose the density $p(x)$ intelligently, the variance of the estimator is reduced. This is called *importance sampling*. $p(x)$ is called the *importance density* and $w_i = \frac{f(X_i)}{p(X_i)}$ is the *importance weight*.

The best possible sampling density is $p^*(x) = cf(x)$ where c is proportionality constant

$$c = \frac{1}{\int_\Omega f(x)dx}. \tag{1.15}$$

Here, the constant ensures that p^* is normalized (i.e., it integrates to 1). The density $p^*(x)$ yields an estimator with zero variance. In practice, we cannot use this density, because we must know the value of the integral we want to compute to evaluate c. However, if we choose an importance density $p(x)$ that has a similar shape to $f(x)$, the variance can be reduced. It is also important to choose an importance density p such that it is simple and efficient to evaluate. In practice, p can be designed by doing some of the following:

1. Discard or approximate some parts of $f(x)$ such that function $g(x) = f(x)p(x)$ can be integrated analytically.
2. Construct a low dimensional discrete approximation of $f(x)$.
3. Approximate $f(x)$ by using Taylor expansion.

After $g(x)$ is designed with any of the above methods, the density is then set to $p(x) \propto g(x)$. We show in next sections how to choose and compute densities in practice.
Note: If the sampling density is not chosen carefully, the variance can be increased and can actually be infinite. Importance sampling is very effective when function $f(x)$ has large values on small portions of the domain. Another common problem that happens in importance sampling is when the sampling density has a similar shape to $f(x)$ except that $f(x)$ has longer (wider) tails. In this case, the variance can become infinite. While importance sampling is a useful and powerful technique it should be used with care. Inappropriate importance density can result in poor estimates of the integral.

Stratified Sampling

If we partition integration domain Ω into a set of m disjoint subspaces $\Omega_1, \ldots, \Omega_m$ (*strata*), we can evaluate the integral as a sum of integrals over the stratum Ω_i. If we generate n_i samples in each stratum (subspace Ω_i), the estimator becomes

$$\hat{I} = \sum_{i=1}^{m} \frac{1}{n_i} \sum_{k=1}^{n_i} f(X_{i,k}) \tag{1.16}$$

whose variance is

$$V(\hat{I}) = \sum_{i=1}^{m} \frac{V_i}{n_i} \tag{1.17}$$

where V_i is the variance of $f(x)$ in stratum Ω_i. The expected error of this method, ***stratified sampling***, is never higher than variance of ordinary unstratified sampling [14]. However, stratified sampling is often better than importance sampling. The two methods can be combined to lower variance even further. Stratified sampling works well for low-dimensional integration, but it does not scale well for integrals of high dimensionality. The number of samples must also be chosen such that there is at least one sample drawn from each stratum.

Control variates

If we can rewrite the estimator as

$$I = \int_{\Omega} g(x)dx + \int_{\Omega} (f(x) - g(x))dx \tag{1.18}$$

where function $g(x)$ can be analytically integrated and has the following property:

$$V[f(x) - g(x)] \leq V[f(x)] \tag{1.19}$$

then a new estimator is

$$F = \int_{\Omega} g(x)dx + \frac{1}{N} \sum_{i=1}^{N} \frac{f(X_i) - g(X_i)}{p(X_i)}. \tag{1.20}$$

The variance of this new estimator will be lower than the original estimator.

How do we decide whether to use importance sampling or control variates? Given function $g(x)$ that is an approximation of $f(x)$, then $g(x)$ can be used either as an importance density or a control variate. If $f(x) - g(x)$ is approximately uniform (constant), then using $g(x)$ as a control variate is more efficient. If $f(x)/g(x)$ is approximately constant, then using importance sampling is more efficient. Note that if g is proportional to p then the two estimators differ only by a constant, and have therefore the same variance. If g is already used as the importance density, it would not be useful as a control variate, because the variance would not be reduced. Another criteria for choosing between importance sampling and control variates is whether $g(x)$ can be integrated analytically (control variates may be preferable) or g can be sampled analytically (importance sampling).

Defensive importance sampling

We already mentioned in Section 1.1.3 that even if a sampling density $p(x)$ has roughly the same shape as a target function $f(x)$, but $f(x)$ has longer tails, importance sampling will fail. When we draw a sample from the tails of $p(x)$, the importance weight can be many times larger than weights in other parts of $p(x)$. This causes high variance and in extreme cases the variance can be infinite. This deficiency can be address with ***defensive importance sampling*** [9] which uses a ***defensive mixture distribution*** $p_\alpha(x)$ instead of only the density $p(x)$:

$$p_\alpha(x) = \alpha q(x) + (1-\alpha)p(x). \tag{1.21}$$

Here, $0 < \alpha < 1$ and $q(x)$ is the target distribution. If we want to compute the integral

$$I = \int_\Omega f(x)q(x)dx \tag{1.22}$$

where $q(x)$ is the target density on the integration domain. The defensive mixture distribution $p_\alpha(x)$ guarantees that the variance is bounded by $1/\alpha$ times the variance of the uniform distribution estimator. It also bounds the importance weight to $1/\alpha$. However, oftentimes it may not be easy or possible to sample from the target density $q(x)$. If $q(x)$ can be decomposed into a product of several (simpler) densities, $q(x) = q_1(x), \dots, q_n(x)$ and each $q_i(x)$ can be easily sampled, then a more general mixture distribution of m densities can be used:

$$p_\alpha = \alpha_0 p(x) + \sum_{j=1}^{m} \alpha_j q_j(x). \tag{1.23}$$

Here, the sum of all weights α_j is one and each weight is greater than zero.

Multiple Importance Sampling

Many times we have to integrate a complex function whose target distribution has multiple modes (peaks or bright regions) and sampling with a single importance density may not capture all regions of the integrand. For example, this is a very common problem in rendering. If our scene contains diffuse and glossy surfaces illuminated by small and large area lights, we face a difficult decision about what sampling strategy to use. For diffuse surfaces, one sampling strategy might be preferable to another. However, the opposite might be true for glossy surfaces.

Suppose that we have n different densities, $p_1(x), \dots, p_n(x)$, and generate n_i samples for each $p_i(x)$. Now, we have many different sampling strategies that work well in different regions of the integrand, but are not good over the entire domain. The question is how to combine multiple strategies that minimize the overall variance without introducing bias. As a naïve approach, one could average the sampling strategies. However, this will not produce optimal results [19].

Instead, we combine all n sampling strategies giving the estimator

$$F = \sum_{i=1}^{n} \frac{1}{n_i} \sum_{j=1}^{n_i} w_i(X_{i,j}) \frac{f(X_{i,j})}{p_i(X_{i,j})} \tag{1.24}$$

where weight $w_i, w_1, \dots, w_n$, provide weight for each sample drawn from some sampling strategy p_i. All weights must be non-zero and the total sum must be 1 to ensure that the estimator remains unbiased. An obvious weighting function would be

$$w_i(x) = c_i \frac{p_i(x)}{q(x)} \tag{1.25}$$

where

$$q(x) = c_1 p_1(x) + \cdots + c_k p_k(x) \tag{1.26}$$

and all coefficients c_i are nonzero and sum to 1.

In general, the best choice of weights turns out to be

$$w_i(x) = \frac{n_i p_i(x)}{\sum_k n_k p_k(x)}. \tag{1.27}$$

If we take exactly one sample, $n_i = 1$ from each sampling density, the weight w_i will be set according to current sampling strategy at x compared to the rest of the strategies:

$$w_i(x) = \frac{n_i p_i(x)}{\sum_k n_k p_k(x)}. \tag{1.28}$$

This weighting strategy is called the *balance heuristic* and is nearly optimal. It is possible to design better strategies for special cases, but universally the balance heuristic out performs most other stratigies.

What is the difference between MIS and Defensive Importance Sampling? Multiple importance sampling (MIS) is optimal for a given set of sampling strategies. However, if we have chosen a bad or inadequate sampling strategy, MIS will not reduce variance. For example, if one of the strategies takes too many samples from low-valued regions and not enough from high-valued regions, the variance will increase. On the other hand, defensive importance sampling (DIS) can improve variance when a sampling strategy p is inadequate. When combined with a uniform density, DIS guarantees that the integrand will be sampled over entire domain. MIS with balance heuristic can be viewed as a special case of DIS (see Equation (1.21), and factor α coming from balance heuristic weights).

We will show in later sections how to use MIS in practice and how to design sampling strategies for rendering.

1.2 Multiple Importance Sampling

Recall that one of the main objectives in rendering is to approximate the illumination integral in Equation (??):

$$L_o(\mathbf{x}, \omega_o) = \int_\Omega L_i(\mathbf{x}, \omega_i) f(\omega_o, \omega_i) \cos\theta_i d\omega_i$$

which can be split into three components: incoming illumination L_i, cosine weighted BRDF B and visibility function V. If we drop the spatial and angular, the illumination integral becomes

$$L_o = \int_\Omega L_i B V d\omega_i \tag{1.29}$$

and the traditional Monte Carlo estimator is

$$\hat{L}_o = \frac{1}{N} \sum_{i=1}^{N} \frac{L_i B V}{p(\omega_i)} \tag{1.30}$$

where $p(\omega_i)$ is the importance sampling density. Ideally, the density function would be proportional to the product of LBV. Unfortunately, this is impossible for all but some artificially contrived scenes. We have to resort to some other density that will hopefully generate low variance in estimate. Let us examine a few possible options.

When we have a diffuse BRDF and multiple area lights of different sizes, we have two obvious choices for importance sampling densities. We can either sample according to the diffuse BRDF or lighting. Figure 1.1 illustrates the two scenarios. Using the diffuse BRDF, we sample the entire hemisphere, but only small portions of the hemisphere contain any lighting. So, many samples are completely wasted since the contribution will be zero. On the other hand, if we sample according to the lighting, none of the samples will be wasted because for any direction in which light is emitted the BRDF will reflect some light. For diffuse surfaces, it is better to sample according to lighting only.

When we have glossy surfaces and many area lights, we can also sample according to the glossy BRDF or lighting densities. Figure 1.2 shows two sampling scenarios. In contrast to diffuse surfaces, glossy surfaces reflect light from a small solid angle. Using light sampling densities, most of the samples will be wasted because the surface will not reflect any light from those directions. Therefore, a better choice is to sample according to the glossy surface reflection, because there is a much larger chance that at least some sampled directions within a reflectance cone will have non-zero lighting contributions.

When we have very glossy surfaces or diffuse only surfaces, the choice of sampling densities is fairly obvious. As highly glossy surfaces become duller (more diffuse) the choice becomes murkier and not straightforward. For slightly glossy surfaces, a combination of two sampling strategies should be used.

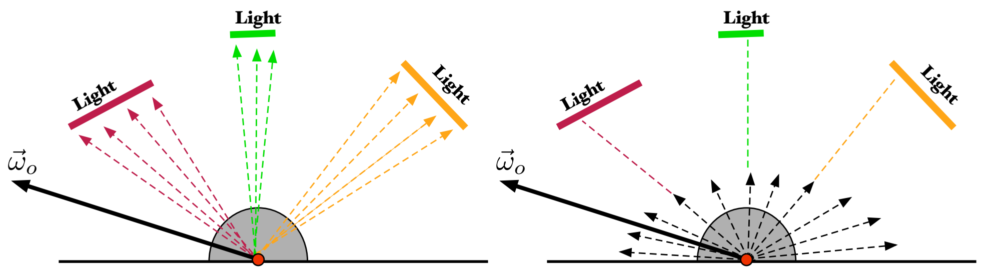

Figure 1.1: Diffuse BRDF and area lights. When we have a diffuse BRDF and multiple area lights of different sizes, two obvious sampling density choices are lighting (left) and BRDF (right). Note that BRDF sampling produces many samples that will be wasted, because there is no light emission in those directions. Sampling according to only the lighting produces lower variance.

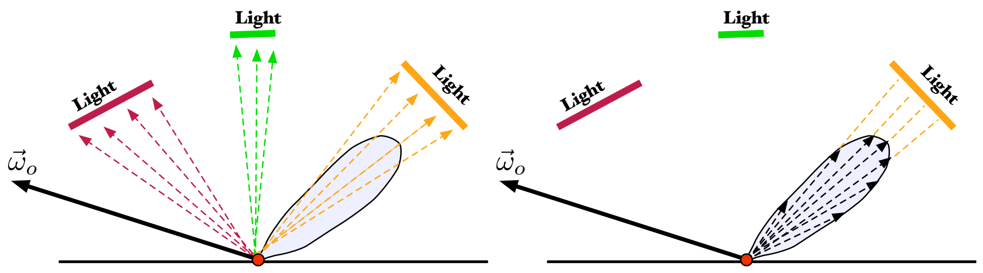

Figure 1.2: Glossy (specular) BRDF and area lights. Lighting (left) or BRDF (right) can be used as an importance sampling density. Note that light sampling produces many wasted samples because there will be no reflection in those directions. Sampling according to the BRDF provides better results.

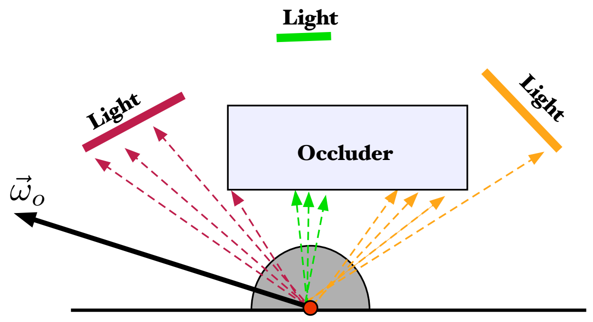

Figure 1.3: Diffuse BRDF and area lights with occluder. While sampling according to lighting is still preferable, the occluder blocks many of the directions that contribute light. Ideally, we would not pick directions that are blocked by the occluder. Unfortunately, this cannot be easily achieved for arbitrary scenes and geometry.

So far, we have only looked at idealized situations where we have simple surfaces (composed of simple BRDFs) and no occluders. This is obviously an unrealistic situation. As shown in Figure 1.3 for a diffuse only surface, once we add occluders to the scene the sampling strategy becomes more complicated. Occluders can prevent light reaching the surface. Sampling according to lighting will generate proper directions, but lighting from those directions might be blocked. For the time being, we ignore visibility in our sampling density but we will return to it later and discuss what we could do to incorporate visibility into the sampling density.

It is clear that complex lighting, surface properties, and occlusions cause the function we want to approximate to be complex and discontinuous. This function can have many bright and dark regions and intensities can differ by orders of magnitude. Since the function is very complex and does not have a nice formulation (due to occlusion) it is clear that either our sampling density will be complex or that we need more than one sampling density.

Veach and Guibas [19] have demonstrated that by combining multiple sampling strategies, the variance can be reduced in situations where a single sampling strategy is bad (see Figure 1.4).

How do we implement Multiple Importance Sampling? Given the two sampling strategies for lighting and BRDF discussed in Section **??**, let $p_1(x)$ and $p_2(x)$ be a BRDF and light sampling density. The random variables X and Y are then

$$\begin{aligned} X_{1,i} &\sim p_1(x) & X_{2,i} &\sim p_2(x) \\ Y_{1,i} &= \frac{f(X_{1,i})}{p_1(X_{1,i})} & Y_{2,i} &= \tfrac{f(X_{2,i})}{p_2(X_{2,i})}. \end{aligned}$$

Now, we just need to combine the samples together:

$$Y_i = w_1 Y_{1,i} + w_2 Y_{2,i}. \tag{1.31}$$

The only remaining question is how to compute weights $w_i(x)$. We have already mentioned a few possible options in Section 1.1. One is using the balance heuristic, where the weights are:

$$w_i(x) = \frac{p_i(x)}{p_1(x) + p_2(x)} \tag{1.32}$$

and the final PDF $p(x)$ for the combined sampling densities is:

$$p(x) = w_1(x)p_1(x) + w_2(x)p_2(x). \tag{1.33}$$

Now, we have all the ingredients to implement multiple importance sampling.

One of the remaining questions is how do we choose the number of samples for each sampling strategy. There are several possibilities:

- Select a fixed number of samples for each strategy. For example, if $N = 100$, then $N_1 = 50$ would be used for lighting sampling and $N_2 = 50$ for BRDF sampling.

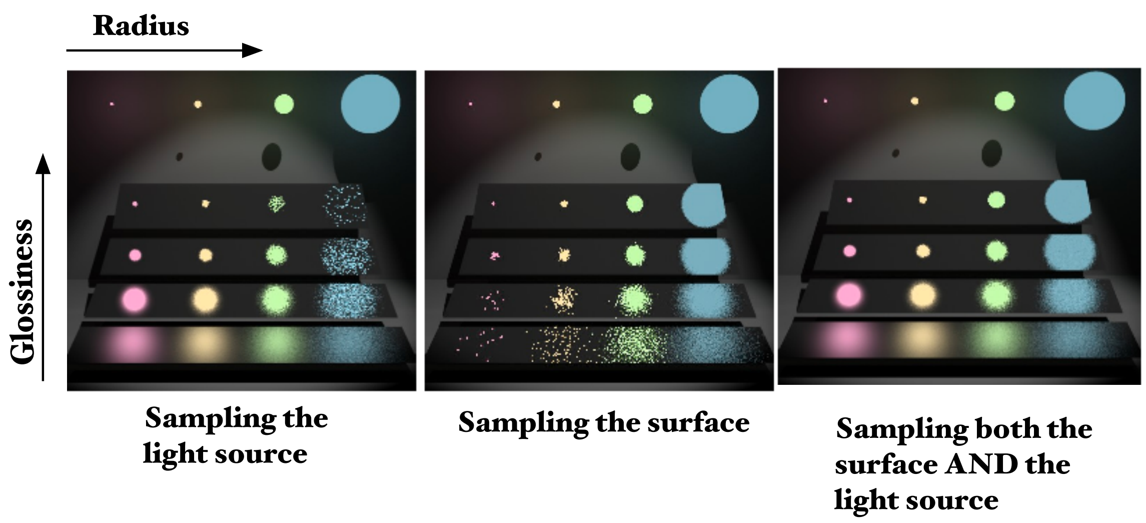
Radius
Glossiness
Sampling the light source
Sampling the surface
Sampling both the surface AND the light source

Figure 1.4: Combining many sampling strategies using Multiple Importance Sampling (MIS) produces superior results to using a single sampling density. Image from Veach and Guibas [19].

Note that this is a relatively safe choice, although it could lead to suboptimal sample generation. If the BRDF is very glossy, some of the samples might be wasted, because too many samples are allocated for lighting sampling.

- Alternatively, the number of samples for each sampling strategy can be adjusted based on a heuristic, such as a combination of the solid angle of the light and glossiness of the surface. A reasonable strategy might be to have a minimum number of samples that will be taken according to each strategy and then distribute the rest based on the heuristic. For instance, if $N = 100$, we might allocate 20 samples to each sampling strategy. The remaining 60 samples would be distributed based on the glossiness of the surface and the light's solid angle.

Notes. Multiple importance sampling is an unbiased method for reducing the variance of Monte Carlo estimators. However, if it is used in conjunction with filtered importance sampling (e.g., filtered importance sampling is used to filter environment lighting) the method is *biased* due to the nature of FIS. For visual effects applications, this is not troublesome as the noise can be greatly reduced.

While MIS reduces the variance, there are still configurations where the variance will be high. As Kollig and Keller [12] pointed out, multiple importance sampling attempts to hide a weakness of using a single density function. If, however, only one sampling density exists for some region of our integration domain Ω, the multiple importance sampling will revert to a standard importance sampling. Kollig and Keller call this an *insufficient set of techniques* [12]. We emphasize again, if inappropriate sampling densities are chosen, multiple importance sampling will not help to reduce the variance.

1.3 Practical Notes on Monte Carlo Sampling

1.3.1 Choosing Sampling Density

The effectiveness of importance sampling depends on the choice of the importance sampling density $p(x)$. Figure 1.5 shows the differences between uniform and importance sampling.

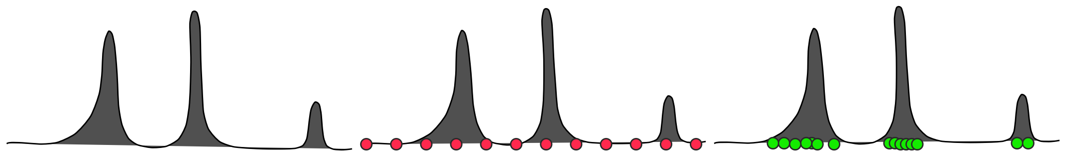

Figure 1.5: (*Left*) A function $f(x)$ can has many *peaks*. There might not be a single importance sampling density $p(x)$ that can capture regions where the function $f(x)$ has large values. (*Middle*) If samples (in red) are chosen uniformly, the variance will be high, because we oversample regions where the function is low (dark regions) and undersamples regions where the function is high (bright regions). (*Right*) If appropriate sampling density is used, we take many more samples (in green) in regions where the function has high values and thus reduce the variance.

Figure 1.6 illustrates why the choice of importance sampling density is crucial for variance reduction. The examples in the figure demonstrate that inappropriate sampling density can increase variance, which can even become infinite.

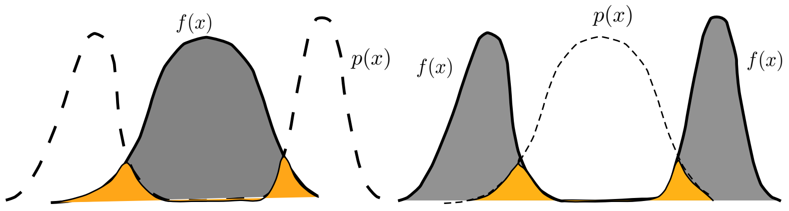

Figure 1.6: Bad choice of importance sampling density. The sampling density does not match the shape of the function $f(x)$ we want to evaluate. Only a small portion of the regions in the density function $p(x)$ overlap (in orange) the non-zero parts of the function. A bad choice of importance sampling density will **increase** variance and not reduce it.

1.3.2 Filtered Importance Sampling For Area Lights

Previous sections have described in great detail how to apply filtered importance sampling for infinite (hemispherical) lights. A small extension could be used for filtered importance sampling for textured area lights.

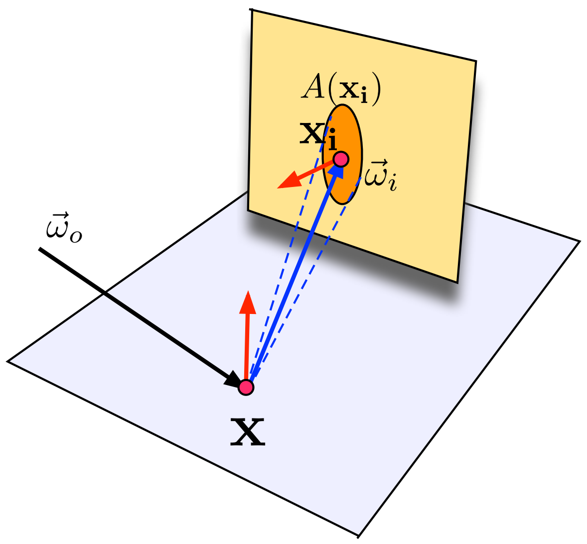

Figure 1.7: Cross-sectional footprint of ray intersection.

Recall that each sampled ray has a solid angle:

$$\Omega_s = \frac{1}{Np(\theta, \phi)}.$$

When the intersection with the sampled ray is found, we can approximate the area on the surface of the hit object by looking at the distance to the hit object and the solid angle of the ray:

$$A(x_i) \approx \frac{\|x_i - x\|^2 \cdot \Omega_s}{\cos\theta_i}. \tag{1.34}$$

For the above to hold true, we assume that the cross-sectional area is locally flat (Figure 1.7). Now that we have the estimated cross-section of the intersection $A(x_i)$, we can estimate the mipmap level l based on this area:

$$l = \frac{1}{2}\log_2\left(\frac{A(x_i)}{A_{\text{pixel}}}\right) = \frac{1}{2}(\log_2 A(x_1) - \log_2 A_{\text{pixel}}) \tag{1.35}$$

where A_{pixel} is the object space area covered by one pixel. It is used to convert areas from object space to texel space. We can use this formula to filter the texture on area lights when using multiple importance sampling. It is important to recognize that this is a crude approximation. When part of the ray footprint is outside the textured area, the light contribution will be underestimated and wrong. Still, this approximation gives plausible results.

1.3.3 What About Visibility?

We have seen in Section 1.1.3 that the idealMonte Carlo estimator should be

$$\hat{L}_o = \frac{1}{N}\sum_{i=1}^{N}\frac{L_i B V}{p(\omega)}.$$

So far, we have been focusing on sampling from either the lighting, BRDF or a combination of sampling strategies using MIS. However, several recent sampling algorithms explicitly compute and sample an approximation of the product between the lighting and BRDF. Two-stage importance sampling [5] and quadtree-based product sampling [4, 3] hierarchically approximate a BRDF at a given point on the surface and combine it with the incoming light L_i. Some of these methods can be fairly costly or may require pre-computed data structures for BRDFs. If BRDFs are spatially varying, some of these methods may not be practical since they would require too much storage for all the BRDFs in the scene.

We can take this further by adding visibility into the mix to lower the variance. Sampling from a triple product of lighting, BRDF and visibility, LBV, is difficult at best. Exact visibility would take a long time to pre-compute and would be expensive to store. We can use an approximate visibility $\bar{V}$, thus making the estimator $LB\bar{V}$. The problem with this approach is that approximate visibility $\bar{V}$ would have to be nonzero everywhere true visibility V is nonzero. This brings us back to the initial problem, because in order to guarantee this condition is satisfied we would have to compute visibility in all directions.

On the other hand, we can use the estimator $LB\bar{V}$ as a control variate:

$$\hat{L}_o = \frac{1}{N}\sum_{i=1}^{N}\frac{L_i B\bar{V} - \alpha L_i B\bar{V}}{p(\omega)} + \alpha\int L_i B\bar{V} d\omega_i. \tag{1.36}$$

Remember that a control variate requires the function $f - g$ to be approximately constant. Even if visibility is crudely approximated, this condition can be satisfied. Clarberg and Akenine Möller [2] analyze the variance for this case and describe an algorithm for using approximate visibility:

- Create a compressed *visibility cache* using a compact bitwise representation stored at a sparse set of points in screen space.
- Compute a rough estimate of $L_i B\bar{V}$ using the approximations.
- Evaluate the difference between this approximation and the correct solution using Monte Carlo integration

The reader is directed to [2] for detailed discussion and implementation notes.

1.3.4 Resampled Importance Sampling

Efficiency of a Monte Carlo estimator depends on the expense of the sample evaluation versus the cost of drawing a sample from better densities. For example, if function is cheap to compute, but finding the importance density is computationally expensive, there is probably an advantage to using a simpler sampling density with more samples. In rendering, casting visibility rays can be expensive and oftentimes we still want to avoid tracing too many shadow rays.

Consider a situation where we have a glossy surface and we choose N samples based on the BRDF density. We also know that the surface is very glossy and therefore the cone around reflected lobe is fairly narrow. We might be able to use less than N visibility rays to approximate the light transport integral. We can do that by using ***resampled importance sampling*** [17]. The idea is that from N partial estimates (BRDF times lighting, L_iB) we only choose M values for which we will compute the visibility.

More formally, resampled importance sampling is a generalization of importance sampling that permits unnormalized sampling densities or difficult to sample densities (in our case, visibility) denoted as q. In rendering, the best density q would be $q \propto L_iBV$, but we can realistically at best only sample from another density p that is proportional to L_iB. Instead of sampling from q, we generate a set of samples from a source distribution p and weight these samples appropriately. Then, we ***resample*** these samples by drawing a single sample from them with probability proportional to its weight.

The basic algorithm proceeds as follows:

- Choose a set of sample X_i from a known distribution p
- Associate a weight $w_i = \frac{q(X_i)}{p(X_i)}$ with each X_i, where q is the desired (possibly unknown distribution)
- Generate the final samples Y_i by sampling X_i with a distribution proportional to w_i

If the weights w_i are chosen to be $w_j = \frac{q(X_i)}{p(X_i)}$ then the resulting samples Y_i will be approximately distributed to q. The processes of resampling is equivalent to filtering. In rendering applications, we use importance resampling as follows:

- Generate N samples from some proposal distribution $p(x)$. This is as before, where we created samples from either lighting density, BRDF density or combined MIS density.
- Compute the weights (e.g., luminance) of partial contributions (L_iB, but no visibility yet).
- Compute the discrete distribution from these weights.

- Chose M samples from the above N samples. These samples are chosen based on the importance density that we computed in previous step.
- Shoot shadow rays for these M samples, add computed visibility to each sample and apply proper weighting (based on the probability with which each sample was chosen).

Note that although the desired target density $q(x)$ is unknown ***a priori*** because of the visibility, we never sample from it. We only need to be able to evaluate it and that is straightforward as long as as we can evaluate visibility V.

Resamples Importance Sampling (RIS) is better than importance sampling when:

- q is a better importance sampling density than p.
- Computing proposals is much cheaper than computing actual samples.

RIS takes advantage of differences in the variance computation expense. More details and examples can be found in [17].

2 Metropolis Light Transport

Metropolis light transport (MLT) is often considered to be the most sophisticated of the unbiased light transport simulation algorithms, with a reputation to be able to render scenes efficiently where other methods fail. To be fair, MLT also has its flaws and much of the efficiency is tightly related to whether the correct mutation strategies for the problem are implemented or not. MLT certainly is not the answer to each and every light transport problem, but its high robustness often make it the tool of choice to render demanding scenes, like architectural shots with pronounced caustics.

A common misconception about MLT is that is generally hard implement. It can certainly get quite complicated, mainly depending on how involved the mutation strategies are and how many are implemented. But then one can also implement a simple MLT variant in very few lines of code. MLT definitely is quite different from classical Monte Carlo rendering approaches and also has somewhat different requirements on what functionality the rendering core needs to provide. The amount of special functionality needed, however, is heavily depending on what flavor of MLT is actually implemented. In general MLT is more like a family of approaches than a single light transport algorithm.

In the following we give a mostly self-contained introduction to MLT, also addressing some implementations issues. The goal is to establish an understanding of how MLT works, when it works best, and what its differences to other Monte Carlo light transport algorithms are.

2.1 Metropolis Sampling

The Metropolis sampling algorithm [13, 8] is a powerful method to generate a process of samples that are, in the limit, distributed according to any target function f. Since the only restriction posed on f is that we need to be able to evaluate it, this provides means to (importance) sample functions that are hard or impossible to handle by other means. We will outline some of the theory and the motivation behind this method without going into to much detail (for a thorough discussion we refer to [10]).

To understand how Metropolis sampling works we look at a stochastic system given on a domain Ω of states and featuring a concept of energy flow between two states. Assume we have such a system and the energy flow from on state x to another state y is governed by a probability density function $K(y|x)$. If we now assume that this system is in equilibrium (i.e. the energy concentration per state does no longer change over time) and that the energy in each state x is expressed by our target function $f(x)$, we have

$$f(x)K(y|x) = f(y)K(x|y). \tag{2.1}$$

This condition, called *detailed balance*, states that the amount of total energy flow from x to y has to be the same as the amount that is flowing into the opposite direction. If this would not be the case, the system would not be in equilibrium.

For some applications such a system would be a real world physical process and $T(y|x)$ would be given by the physical laws driving that process. In our case the system is merely a mathematical construct and we do not directly know what a suitable $K(y|x)$ for the target function $f(x)$ would to look like. The key idea now is to replace $K(y|x)$ by an almost arbitrary distribution $T(y|x)$ and only accept a movement from x to y with probability $a(y|x)$. This means that we have set $K(y|x) = a(y|x)T(y|x)$ and detailed balance is now expressed as

$$f(x)a(y|x)T(y|x) = f(y)a(x|y)T(x|y)$$

which leads to the relation

$$\frac{a(y|x)}{a(x|y)} = \frac{f(y)T(x|y)}{f(x)T(y|x)}$$

for the acceptance probability. We can fulfill this relation by setting

$$a(y|x) := \min\left\{\frac{f(y)T(x|y)}{f(x)T(y|x)}, 1\right\}. \tag{2.2}$$

The stochastic system defined by $a(y|x)T(y|x)$ is a Markov chain, where the transition from the current state x to the next state x' is based on a proposal y (chosen with probability density $T(y|x)$) and an acceptance probability for that proposal, i.e we set $x' = y$ with probability $a(y|x)$ and $x' = x$ with probability $1 - a(y|x)$. Since we have constructed $a(y|x)$ such that we obtain detailed balance for the target function $f(x)$, the chain's stationary distribution $p(x)$ is proportional to the target function, i.e. $p(x) = \frac{f(x)}{b}$ where the normalization $b = \int_\Omega f(x)dx$ does not need to be known for the method to work.

If we now simulate a trajectory of this Markov chain we know that once it has reached the stationary distribution its states are distributed according to the target function $f(x)$. Simulating the Markov chain thus provides the means to draw samples from f: this is the Metropolis sampling algorithm. Let us summarize what we need to implement it.

- We need to be able to evaluate the target function f.
- We need to implement a *mutation strategy* that given a state x creates a tentative state y according to a probability density $T(y|x)$. In order to ensure the ergodicity of the resulting Markov chain we need to have $T(y|x) > 0$ whenever $f(x) > 0$ and $f(y) > 0$. This also ensures that equation 2.2 is always well defined for all proposals we create.

- In order to compute the acceptance probability we need to be able to compute both $T(y|x)$ and $T(x|y)$ (ideally they are equal and cancel in equation 2.2).
- We can only use the states produced by the simulation process after the stationary distribution has been reached, thus we need to discard a certain amount of samples in the beginning. This issue, known as start-up bias, is generally a hard one but can be avoided or compensated under certain circumstances.

Apart from the points we just mentioned there is a lot of freedom to construct the mutations – which is one of the key strengths of the algorithm. Often a Gaussian along each coordinate axis (and centered in the current state) is used as mutation. This favors small step sizes but still ensures that the whole domain is covered. However, there is little restriction on what the mutation strategy has to look like. In fact it can be hand tailored to the problem that is faced, e.g. we can construct mutations targeted to explore certain lighting effects. Also we do not need to restrict ourselves to a single mutation strategy, we can randomly choose from a set of possible mutations $T_1, \ldots, T_n$. If we choose T_i with probability p_i (where $\sum_{i=1}^{n} p_i = 1$), then we have the probability density of the combined strategy as $T(y|x) = \sum_{i=0}^{n} p_i T_i(y|x)$.

2.2 Application to Light Transport

Photorealistic image synthesis can be seperated into two parts, computing the radiance function throughout the scene (and how it arrives in the camera model) and then projecting that function onto a discretized two dimensional image, i.e. the grid of pixels. The projection is defined by a pixel filter function w_j and the value of pixel j is given as

$$v_j = \int_{\Omega} f(x) w_j(x) dx. \tag{2.3}$$

Here Ω is the domain where the light transport paths are defined on and f gives the measurement contribution of such a path. The pixel filter function w_j is defined on the image plane (which is a two dimensional subset of the whole integration domain Ω) and usually has a small support.

Estimating the radiance function and projecting it to a pixel are often tightly connected in an implementation since it feels natural to work on a per pixel basis. A typical example is a forward path tracer that creates the image pixel per pixel by shooting rays through that pixel's filter's support. However, we can also view this as an integration problem over the whole image plane, where rays are started from random positions on the image plane and just those pixels' values are influenced where w_j is non-zero. Like this a forward path tracer that works per pixel can interpreted as sampling that works on the whole image plane and employs stratisfication such that each pixel receives the same number of samples.

In the following we will work on the whole image plane and not compute integrals pixel by pixel. Of course we still compute the integral v_j per pixel, but we do this by sampling x using some density $p(x)$ on the whole domain Ω, and thus on the whole image plane. (Usually the first two dimensions of Ω will be the image plane and the $p(x)$ is a product density where a uniform distribution is used for those first two dimensions.) For $x_1, x_2, \ldots$ sampled according to $p(x)$ we now can compute $f(x_i)/p(x_i)$, and $w_j(x_i)$ for all pixels (for most of which $w_j(x_i)$ will be zero), yielding a Monte Carlo estimator that computes all pixel integrals:

$$v_j = \int_\Omega f(x) w_j(x) dx = E\left[\frac{1}{N}\sum_{i=1}^{N} \frac{f(x_i)}{p(x_i)} \cdot w_j(x_i)\right].$$

Now it is just one more step to MLT. Instead of using the density p we use the Metropolis sampling algorithm with target function f to create samples $x_1, x_2, \ldots$. If we assume we have already reached the stationary distribution then those samples are distributed according to f and we have

$$v_j = \int_\Omega f(x) w_j(x) dx = E\left[\frac{1}{N}\sum_{i=1}^{N} b \cdot w_j(x_i)\right].$$

The samples from the Markov chain simulation "randomly move around" the image plane, with a distribution proportional to the radiance arriving on the image plane. As such we just need to accumulate a histogram per pixel, where we add up $b \cdot w_j(x)$ for each Markov chain sample falling into the support of the pixel. This is MLT:

1. Estimate normalization constant, e.g. by using a Monte Carlo estimator

$$b = \int_\Omega f(x) dx \approx \frac{1}{N}\sum_{i=1}^{N} f(x_i)/p(x_i). \tag{2.4}$$

2. Choose a initial sample x_0, set $x = x_0$.

3. For $i = 1$ to M do:
    - y = mutate x (according to $T(y|x)$)
    - compute $a = \min\left\{\frac{f(y)T(x|y)}{f(x)T(y|x)}, 1\right\}$
    - set $x = y$ with probability a
    - for all pixels j with $w_j(x) > 0$, add value $\frac{w_j(x) \cdot b}{M}$

Strictly speaking, step 1 can already be a hard problem. However, let us put this into perspective by comparing it to the problem of computing an actual image. Think about computing a 1000×1000 image with one sample per pixel: this already translates to having thrown one million samples at equation 2.4. And even if the normalization constant is off by small amount, this basically only means that the resulting image will just be a bit too dark or a bit too bright. Unless we want to render an animation this might not be as important after all.

Using the Expected Value for Contributions We will accept the proposal y from the current state x with probability $a(y|x)$, so we will add a contribution $b \cdot w_j(y)$ for state y (i.e. to all pixels j with $w_j(y) > 0$) with probability $a(y|x)$ and a contribution $b \cdot w_j(x)$ for state x with probability $1 - a(y|x)$. Instead of randomly adding the contribution for one state or the other, we simply add the expected value of contribution for both states, which means we add $b \cdot w_j(y) \cdot a(y|x)$ for state y and $b \cdot w_j(x) \cdot (1 - a(y|x))$ for state x.

Colors The outlined MLT algorithm assumes that $f(x)$ is a scalar function and thus computes a scalar output image. Since we usually are interested in color output we need to apply some minor modification. For a vector-valued $f(x)$ we use an arbitrary intensity function l that transforms those vectors to scalars, and then use the scalar-valued $f_l(x) := l\,(f(x))$ as target function.

Instead of recording equal valued contributions of magnitude b on the image plane we now record $b_l \cdot f(x)/l(f(x))$ where b_l is the normalization constant resulting for the scalar-valued target function f_l.

- A typical choice for l is the CIE luminance of the computed color. The intention here is to have the distribution follow the response characteristics of the human eye. However, sometimes is better to be more conservative and not to undersample colors that have lower CIE luminance, e.g. by using the maximum of all color channels as l.
- For full spectral simulation, the wavelength is usually one more dimension of the problem and can be sampled by the Metropolis sampling algorithm along with all the other components. Here the result is an intensity value and the sampled wavelength. Accumulation then usually takes place in a tristimulus color space, such as CIE XYZ, where the spectral values are converted to. So we could again use the CIE luminance or, more conservatively, just the spectral intensity value (without further wavelength dependent weighting) as l.

Start-Up Bias The Markov chain created by Metropolis sampling has the desired stationary distribution but it can take a while until the simulation process reaches that distribution. The intuitive understanding here is that the process is biased towards a starting state which needs to be “forgotten”. Unfortunately, it is impossible to tell to when this is the case, it is depending on how correlated successive samples are and how many have been taken.

- If we simulate a single Markov chain start-up bias can mostly be ignored. As long as the initial state has a non-zero contribution (and thus a non-zero probability to be created by the Markov chain) it does not really matter where we start since the state could have been chosen anyways (albeit at low probability).

- If many Markov chains are simulated in parallel all of the chains are biased towards the distribution used to sample the initial states. Since we can typically spent less samples per chain if we simulate many chains, this will influence the result for quite a while.
- Start-up bias obviously would not be an issue if we could already choose the starting state according to the stationary distribution. This, of course, is something we probably cannot do – otherwise there would be no need to apply Metropolis sampling in the first place. However, we can approximately do it using re-sampling from a larger set (see next item).
- If the initial state x_0 is chosen using a density $p(x_0)$ and if we use $f(x_0)/p(x_o)$ as weight for all samples created by the chain (instead of b), the estimator is unbiased [20, 18], i.e. the expected value is the correct integral. Of course, this is not directly useful if applied to a single Markov chain, since it only leads to the correct image brightness in expectation, i.e. when many chains weighted like this are averaged.

  Better is to sample a large number n_0 of initial states $x_{0,j}$ according to $p(x_{0,j})$ for $j = 1, \ldots, n_0$ (those samples can also be used to estimate the normalization constant b). Then we re-sample a smaller set of $n \ll n_0$ chains (or even a single chain) based on the values $w(x_{0,j}) = f(x_{0,j})/p(x_{0,j})$ and use an equal weight for the re-sampled chains [18].

2.2.1 Efficiency

Metropolis sampling follows the targeted function exactly which, from a Monte Carlo simulation point of view, is as good as we can get. Of course this property does not come for free: the samples are highly correlated. We can differentiate between two sources of correlation.

- Correlation due to rejection: if the probability of acceptance for proposals is low, we can get stuck in one state for a very long time.
- Correlation due to small mutations: the smaller the mutation, the more similar successive states are.

Ideally, we would have a mutation strategy were the proposals are fully independent and are accepted with probability one. It is pretty clear that this would mean that we could already perfectly importance sample the target function by other means. In fact it boils down to exactly this: setting the acceptance probability in equation 2.2 to one and having independent mutations, i.e. $T(b|a) = T(b)$, we have $f(y)/f(x) = T(y)/T(x)$ which implies that f and T can only differ by scaling, so T is already perfectly importance sampling f.

Similarly as a standard path tracer can suffer from high variance problems when a highly contributing effect is sampled at low probability, the samples from MLT can yield high variance if the mutations are bad in finding significant regions of the integrand and have low acceptance probability. It is clear that the key to high efficiency is good mutations strategies that keep the variance low. Asymptotically, however, the behavior of MLT is the same as for uncorrelated standard Monte Carlo path tracing, the variance per pixel is decreasing linearly in the number of samples taken [1], so we do not have a gain or loss from using correlated samples in general.

A certain drawback of the Metropolis sampling approach is that stratisfication, especially across the image plane, cannot be obtained. There is no control to influence into what pixel a sample should fall to and the resulting variance in the amount of how many samples a pixel has received at a certain time leads to more low frequency noise – even for simplest scenes.

2.3 Mutation Strategies

The Metropolis sampling framework leaves a great degree of freedom to what the mutations that create a proposal state y from a current state x can look like. In practice there are basically two approaches for implementing mutations, one is to work on the sample numbers directly, the other to explicitly modify light transport paths.

2.4 Primary Sample Space Mutations

The straight forward application of Metropolis sampling to light transport is to just mutate the random numbers (in the infinitely dimensional unit cube I) that are used to drive an unbiased Monte Carlo light transport estimator [11]. This estimator, let us call it primary estimator, can be any kind of Monte Carlo light transport algorithm, e.g. a path tracer, a light tracer, or a bidirectional path tracer. Fed by a uniform random sample $x \in I$ it creates a light transport path (or a set of light transport paths) with a certain contribution, e.g. for a simple path tracer $t(x) = f(x)/p(x)$. Although in expectation the contribution of $t(x)$ recorded on the image plane is exactly what we want to compute, the efficiency of the primary sampler might be low because the probability density $p(x)$ just does not match some parts of the integrand well enough. In practice these are the parts were the primary sampler does not employ a suitable importance sampling technique, like caustics in the case of a simple path tracer. Even with the many techniques at hand with a bidirectional path tracer, there might be a high contribution effect that is just not handled by any of them.

In order to address the problems of the primary sampler we can wrap it up in Metropolis sampler: we mutate the random sample x according to some probability density function on I. Generally two types of distribution are used:

- A large step mutation proposes a (fully independent) new random sample, drawn from a uniform distribution. Here the proposal state does not depend on the current state, i.e. $T_{\text{large}}(y|x) = T_{\text{large}}(y) = 1$.

- A small step mutation adds a small offset (following some convenient distribution) to the current sample. Usually, the distribution is constructed such that $T_{\text{small}}(y|x) = T_{\text{small}}(x|y)$, e.g. by sampling an independent and symmetric offset per component.

We now randomly decide whether to use a large step mutation with probability p_{large} or a small step mutation with probability $1 - p_{\text{large}}$. Then we have

$$T(y|x) = p_{\text{large}} + p_{\text{small}} \cdot T_{\text{small}}(y|x).$$

Note that T is symmetric if T_{small} is symmetric, in which case we do not have to compute it since it cancels out in equation 2.2.

Implementing Small Steps Small step mutations can be implemented by applying a small offset to the current sample number, where we have to keep in mind that smaller steps are generally more likely to be accepted but yield higher correlation. A typical choice is to apply a random, preferably small, symmetric offset per component, i.e. if $x = (x_1, x_2, \dots)$ we add an independent offset per x_i. The proposal in [11] is to use

$$(y_1, y_2, \dots) = (x_1 + \text{smallstep}(x_1), x_2 + \text{smallstep}(x_2), \dots)$$

where

$$\text{smallstep}(x_i) = \pm s_2 e^{-\ln(s_2/s_1)\cdot\xi_i}$$

for ξ_i uniformly distributed in $[0, 1]$ and the sign chosen at random. This results in a minimum step size of s_1 and maximum step size of s_2 with higher probabilities for smaller steps. The recommendation from [11] is to use $s_1 = 1/1024$ and $s_2 = 1/64$.

Multiple Importance Sampling for Large Steps Besides being a valid mutation strategy for the Metropolis sampler, large step mutations could also form a standard Monte Carlo estimator. In fact that standard Monte Carlo estimator can yield higher efficiency for some regions of the integrand, essentially those regions which it oversamples. Since we have computed them anyways, it can thus be useful to interpret large step mutations as both samples of the Metropolis sampler and a standard Monte Carlo sampler [11]. Then we can combine the samples from the two techniques in a low variance fashion using multiple importance sampling [19].

In the terms of multiple importance sampling we have two techniques, the first technique is Metropolis sampling and its probability density p_1 (defined on the infinite dimensional unit cube) is proportional to the target function $t(x) = f(x)/p(x)$, i.e. we have $p_1(x) = \frac{1}{b}t(x)$. The second technique is Monte Carlo sampling formed by large steps, which are uniformly sampled and created with probability p_{large}, i.e. we have $p_2(x) = p_{\text{large}} \cdot 1$.

- The current state x is a sample from Metropolis sampling, using the balance heuristic we get a multiple importance sampling weight of $p_1/(p_1 + p_2)$. Combining it with the usage of the expected value, the contribution we add for the current state is

$$(1 - a(y|x)) \cdot \frac{p_1(x)}{p_1(x) + p_2(x)} \cdot \frac{t(x)}{p_1(x)} = (1 - a(y|x)) \cdot \frac{t(x)}{t(x)/b + p_{\text{large}}}.$$

- Analogously, the proposed state y is a sample from Metropolis sampling, where it gets contribution

$$a(y|x) \cdot \frac{p_1(y)}{p_1(y) + p_2(y)} \cdot \frac{t(y)}{p_1(y)} = a(y|x) \cdot \frac{t(y)}{t(y)/b + p_{\text{large}}}.$$

  If the proposed state results from a large step mutation, it is also a sample from Monte Carlo technique and we have the additional contribution

$$\frac{p_2(y)}{p_1(y) + p_2(y)} \cdot \frac{t(y)}{p_2(y)} = \frac{t(y)}{t(y)/b + p_{\text{large}}}.$$

As desired the weighting now favors the Monte Carlo estimator over the Metropolis estimator for large steps if $t(y)$ is small and unlikely to be accepted. The higher p_{large} is, the more Monte Carlo samples we produce and the more they influence the weight for the Metropolis samples.

Note that it is not necessary to have the exact value of b since multiple importance sampling does not require p_1 to be the exact probability density in order to be unbiased. However, it should be a good approximation, such that the balance heuristic works well. Of course we could also use the power or maximum heuristics.

Estimation of the Normalization Constant We can make further use of the fact that large step mutations compute valid samples for the primary estimator: by summing up $t(x)$ we can progressively refine the normalization constant b. In fact, unless there is a need for early feedback or we want to employ multiple importance sampling for large step mutations, we can completely skip the step of pre-estimating the normalization constant and only compute it on the fly using large steps.

Large Step Probability Essentially, we need to balance large step mutations, which reduce correlation and can help to move out of local maxima, against small step mutations, which explore the space more locally and are usually more likely to be accepted. Furthermore, we need to take into account that large steps can have additional value, such as to compute the normalization constant and (weighted by multiple importance sampling) as direct contribution for darker images areas. As there is no general best way to set p_{large}, a conservative strategy is to not set it too high and not set it too low. A relatively robust choice is to use a value between 0.1 and 0.5.

Lazy Evaluation of Mutations Although in theory light transport is a infinitely dimensional integration problem, in practice only a limited number of light bounces are simulated, each of them usually consuming a limited amount of pseudo-random numbers. Even if the primary sampler employs Russian roulette a real world implementation should limit the amount of bounces in order to avoid unpredictable runtime or even infinite loops due to numerical imprecision (in floating point world path tracers tend to get stuck somewhere).

So let us assume that the maximum amount of sample numbers drawn for each path space sample of the primary estimator is limited. We can now store both the current state and the mutation state in a fixed size array of that maximum size. A naïve implementation would loop over all samples of the current state, apply a mutation to that number, and store it in the mutation state. Then, if the mutation state is accepted, we overwrite the current state with the mutation state, otherwise we keep it unchanged.

In practice, however, a path will likely terminate (much) earlier and not all dimensions are used. For efficiency reasons it is therefore desirable that we only have to compute the mutations that are actually required.

For large steps mutations this is very simple, here we do not need to take the current state into account and just create a new sample number dimension by dimension as needed. Small steps on the other hand need the previous state, so in case the previous path used less dimensions we might need to replay the history of previous mutations up to the current point in time. Of course we only have to do this starting from the last accepted large step, since that one is independent of any previous history. This is the implementation proposed in [11], where a time stamp is stored alongside each dimension, to keep track of how many small steps need to be performed since the last large step. A drawback is that sometimes this only postpones the effort: if a path is much longer then the previous one and it has been a while since the last large step was accepted the sampling gets very costly.

So let us apply a small trick to simplify this a bit more. For dimensions that were not used in the previous state we just always apply a large step. This means we formally define the probability density for a small step proposal of the i-th component as

$$T^{(i)}(y_i|x_i) = \begin{cases} 1, & \text{component } i \text{ is used in } f(x) \text{ or } f(y) \\ \text{smallstep}(x_i), & \text{otherwise.} \end{cases}$$

Now we have $T(y|x) = \prod_{i=1}^{m} T^{(i)}(y_i|x_i)$, where m is the minimum of the maxima of dimensions up to which we use the samples in $f(x)$ and $f(y)$. In fact we do not really need to care about that if we move from a longer to shorter path, because then we do not create the sample in the first place. However, by defining it like this we conveniently have $T(y|x) = T(x|y)$. Now we only need to store the maximum dimension we have actually used along with the state and only need to check against that whether to do a small or large step for the current component. An additional benefit is that we only need to back up an accepted state up to that number, potentially copying much less data. The resulting sampler implementation is quite compact, listing 2.1 contains the code in C.

Mapping of Dimensions One key assumption for applying small mutations on the pseudo-random numbers directly is that small changes there translate to small changes on the resulting light transport path. For most cases this true, since the implementations of importance sampling employed by the primary estimators typically are rather smooth functions. However, there are usually also random choices involved, like choosing between BSDF components, that can cause drastic changes for all subsequent vertices. To a certain extent this unavoidable, but we can try to minimize the impact by mapping the dimensions from the unit cube to the problem carefully.

- A bidirectional path tracer creates both a light and eye path by sampling a particle trajectory through the scene. Each of them is using a certain number of pseudo-random samples, which in the case of Russian roulette might not be known initially. Now in order to have small changes on the sample result in small changes on the actual path it is important that the same dimensions are used for the same path segment. We absolutely should avoid that the length of the light path changes the dimensions used for the eye path (and vice versa). A popular way to do this is to use odd dimensions for one path type and even ones for other.

  Alternatively, we can just have two separate arrays of current sample points, one is used for the light path and one is used for the eye path. This is also useful for a lazy sampler implementation, since we can apply the trick to always do a large step mutation in case the current dimension was not used for the previous sample separately per path type.

- Sampling the next path vertex from the existing one might need a different amount of pseudo-random samples depending on the material model. On the extreme side we have perfectly specular reflection which does not consume any samples, while a typical multi-component BSDF model usually consumes three. As a consequence, a small mutation might suddenly change subsequent vertices radically, just because a different material is encountered, e.g. a direct light sample chooses a different light source. Thus it often pays off to always draw the maximum number of samples per bounce even if they are not used (especially since the cost of generating a sample is negligible compared to cost of ray tracing and material evaluation).

- Sometimes the number of samples needed to create the next vertex cannot be bounded. The typical example here is the usage of rejection sampling to importance sample the BSDF which, fortunately, is seldom used in practice since most BSDFs offer more convenient means to be sampled. Another example is using Woodcock tracking to importance sample the transmittance function of heterogeneous participating media [21, 6], a key element for solving volume light transport in an unbiased way.

```
typedef struct {
    /* previous state */
    double samples_prev[MAX_DIM];
    int max_dim_prev;
    /* current mutation */
    double samples[MAX_DIM];
    int current_dim;
    bool large_step;
} state_t;

void init (state_t *state)
{
    state->max_dim_prev = 0; /* no history yet */
    state->current_dim = 0; /* start */
}

double sample(state_t *state)
{
    /* decide between small and large step mutation */
    if (state->current_dim == 0)
        state->large_step = (rand01() < LARGE_STEP_PROB);

    /* also do a large step mutation if the previous state has used less components */
    if (state->large_step ||
        state->current_dim >= state->max_dim_prev)
        samples[state->current_dim] = rand01();
    else
        samples[state->current_dim] =
            smallstep(state->samples_prev[state->current_dim]);

    return samples[state->current_dim++]; /* increment current dimension */
}

void accept(state_t *state)
{
    /* copy back used dimensions */
    for (int i = 0; i < current_dim; ++i)
        state->samples_prev[i] = state->samples[i];
    state->max_dim_prev = state->current_dim;
    state->current_dim = 0; /* proceed */
}

void reject(state_t *state)
{
    state->current_dim = 0; /* proceed */
}
```

Listing 2.1: Primary sample space lazy Metropolis Sampler

Let us assume we use Woodcock tracking to sample the next interaction vertex, which might either be a point in the volume or the intersection point with the next surface along the ray. If the volume is thin along the current path, chances are that we have used a high number of sample points for Woodcock tracking and still reach the same surface. Here we ideally want to use mutations of the numbers we used for that vertex in the current state and nothing something that depends on the number of Woodcock tracking steps we did in the volume. However, for the Woodcock tracking process we of course also want to use mutations of the numbers that were used for the Woodcock tracking process before, since it might very well be that the volume is rather thick and the interesting effect we want to explore is a volume effect. In order to obtain both, we need to reserve the dimensions in a two dimensional fashion, e.g. an array of arrays, where each array is associated to one bounce and holds a fixed number of points for BSDF or phase function sampling, and an arbitrary number of points for Woodcock tracking [15].

In an implementation the arrays of course should be of finite size, but even as such the memory requirements for storing the state are increased drastically.

2.5 Path Manipulation

The original formulation of Metropolis light transport [20] constructs the mutations directly in path space. This has both advantages and drawbacks: while it allows to manipulate existing paths very locally and focused on very specific effects, it is no longer an orthogonal technique that can just be applied to an already existing primary light transport estimator. Here the mutations themselves shape the rendering core and more care needs to be taken in order to be able to compute $T(y|x)$.

We can basically differentiate between two mutation strategies, so called bidirectional mutations that create new paths from old by deleting or adding vertices, and pertubations, which change the positions of already existing vertices slightly.

2.5.1 Bidirectional Mutations

Let us assume we have a light transport path $x = x_0 \dots x_k$ with $k + 1$ vertices, where the first vertex x_0 is on a light source and last vertex x_k is on the lens. A bidirectional mutation now constructs a new path y by replacing a subpath of x with a new subpath (of possibly different length).

In order to compute $a(y|x)$ we need to compute $f(x)$, $f(y)$, $T(y|x)$, and $T(x|y)$. The target function is the contribution function of the path, i.e. if we have a path $x = x_0 x_1 \dots x_k$, then

$$f(x) = L_e(x_0 \to x_1) G(x_0 \leftrightarrow x_1) \prod_{i=1}^{k-1} \left(f_s(x_{i-1} \to x_i \to x_{i+1}) G(x_i \leftrightarrow x_{i+1})\right) W(x_{k-1} \to x_k)$$

We quickly recall the terms used in above equation, for details please refer to [18],

- L_e characterizes the radiance emitted on the light source,
- $f(x \to y \to z)$ is the BSDF for light coming from point x that is scattered at point y into the direction of point z,
- $G(x \leftrightarrow y) = \frac{|\cos\theta_x||\cos\theta_y|}{\|y-x\|^2} V(x \leftrightarrow y)$ is the geometric term, the product of the cosines at the points x and y over the squared distance times the binary visibility between those points, and
- W is the response of the camera model.

In a standard Monte Carlo estimator $f(x)$ is computed alongside a probability density $p(x)$ used to create the sample x which yields the contribution of the sample, $f(x)/p(x)$. In contrast to that, for MLT we are interested in the ratio defining the acceptance probability 2.2. Therefore note that we do not have to compute the parts of f that are shared between the paths x and y, since the cancel out in that equation. But now for the steps that create a new path from the current path:

- We choose to delete the subpath $x_s \ldots x_t$ (not including the vertices x_s and x_t) where $-1 \leq s < t \leq k+1$ is chosen with probability $p_{\text{delete}}(s,t)$. Note that there is the possibility that the current path is disposed completely.
- We extend the remaining path segments by starting random particle walks from their end points. The light path $x_0 \ldots x_s$ is extended by s' vertices, $l_1 \ldots l_{s'}$, at its beginning, the eye path $x_t \ldots x_k$ is extended by t' vertices, $e_1 \ldots e_{t'}$, at its end.

  The two paths are then joined by connecting the endpoints, resulting in the mutated path $y = x_0 \ldots x_s l_1 \ldots l_{s'} e_{t'} \ldots e_1 x_t \ldots x_k =: x_0 \ldots x_s z_1 \ldots z_{s'+t'} x_t \ldots x_k$.
- For the light path we sample the BDSF at x_s and trace a ray to determine the next vertex x_{s+1}. This process is continued until we have s' new vertices. In case the light path is empty initially, we need to sample a position on the light sources and start the random walk by sampling the EDF there.
- Analogously we sample t' new vertices starting from vertex x_t for the eye path. If the eye path is empty initially, we need to sample a new pixel position and a point on the lens to obtain the first vertex and direction.
- We now can describe f for the parts of the path that differ between x and y. For x this is

$$f'(x) = \prod_{i=s}^{t-1} \left(f_s(x_{i-1} \to x_i \to x_{i+1}) G(x_i \leftrightarrow x_{i+1})\right) f_s(x_{t-1} \to x_t \to x_{t+1}),$$

for y we have

$$\begin{aligned} f'(y) =& f_s(x_{s-1} \rightarrow x_s \rightarrow z_1)G(x_s \leftrightarrow z_1)f_s(x_s \rightarrow z_1 \rightarrow z_2)G(z_1 \leftrightarrow z_2) \\ & \prod_{i=2}^{s'+t'+1} (f(z_{i-1} \rightarrow z_i \rightarrow z_{i+1})G(z_i \leftrightarrow z_{i+1})) \\ & f(z_{s'+t'-1} \rightarrow z_{s'+t'} \rightarrow x_t)G(z_{s'+t'} \leftrightarrow x_t)f(z_{s'+t'} \rightarrow x_t \rightarrow x_{t+1}). \end{aligned}$$

Note that for simplicity we have ignored the special cases at the path ends where the emission or the camera response come into play.

- In order to obtain the probability density, we need to look at the probability density involved in implementing the change. All probability densities are expressed with respect to the area measure.

  This is best illustrated by an example: assume we would have a mutation strategy that either shortens the path with probability q by the last vertex or extends it there with probability $1-q$. Now assume we have added one vertex x_{n+1} to a path $x_1 \ldots x_n$ by sampling the BSDF at vertex x_n: If we use a density on the projected hemisphere p_s to do that and x_{n+1} is the resulting next surface intersection point, we have $T(x_1 \ldots x_n x_{n+1} | x_1 \ldots x_n) = (1-q) \cdot p_s(x_{n-1} \rightarrow x_n \rightarrow x_{n+1}) \cdot G(x_n \leftrightarrow x_{n+1})$. For the reverse direction we have $T(x_1 \ldots x_n | x_1 \ldots x_n x_{n+1}) = q$.

  Similarly, we can now compute $T(y|x)$ for the subpath we have inserted, and $T(x|y)$ for the subpath that was deleted. Note that it is important to include all possible ways a given subpath with l vertices was created in T, i.e. all combinations of s' and t' with $s' + t' = l$ need to be taken into account. More details will be included in an updated version of the course notes.

Ergodicity Bidirectional mutations fulfill the ergodicity requirement for the Markov chain, i.e. $T(y|x) > 0$ whenever $f(x) > 0$ and $f(y) > 0$. In other words we can move from any contributing state to any other contributing state. A consequence is that we (theoretically) always have to compute $T_{\text{bidirectional}}(y|x)$ (and $T_{\text{bidirectional}}(x|y)$) and include that in $T(y|x)$ (and $T(x|y)$) for any other mutation strategy, since it could have been constructed by a bidirectional mutation as well.

2.5.2 Pertubations

Pertubations are modifying the positions of (some of the) vertices of the current path, usually by slightly modifying the direction from one vertex to next, while leaving the path topology intact. They are intended to yield high acceptance probability by only do small chances, targeted at very specific lighting effects. A positive side effect of pertubations is that the implementation is often simpler and more efficient as compared to the more general case of bidirectional mutations. More details on pertubations will be included in an updated version of the course notes

2.6 Two-Stage MLT

A direct consequence of importance sampling of the target function, which in expectation is the intensity arriving on the image plane, is that brighter areas receive more samples that darker areas do. More precisely, the ratio of the number of samples arriving in a pixel to a given total number of samples is proportional to the brightness of that pixel. This is not really desirable, since it leads to more perceivable noise in dark image regions whereas bright details might receive more samples than a (most likely tonemapped) final image would need.

- The problem is most prominent if very bright light sources are directly visible, as an extreme example think about looking at the daylight sky with the sun in the field of view. Even though the sun only subtends a very small solid angle, it directly contributes most of the intensity of daylight. Thus, the majority of samples produced by MLT will be concentrated in the (probably very few) pixels showing the sun.

- The most exteme cases can be avoided by excluding directly visible light sources, or even direct lighting [20], from MLT since they usually can be computed robustly by other means. The obvious drawback is that we then have an inhomogeneous solution, where we need to decide how much computation to spent on each subproblem.

- Ideally we would want to have approximately the same (expected) number of samples per pixel and still importance sample the target function within that pixel, i.e. the intensity arriving within that pixel.

  Assume that we would know the final intensity of a pixel v_j, then we could change the target function $t(x)$ using that intensity to

  $$t'(x) = \frac{t(x)}{\sum\limits_{\text{pixels } j : w_j(x) > 0} w_j(x) v_j}.$$

  This would lead to equal probabilities for all pixels to be sampled. We would obtain the final image then by multiplying the result (an equal valued image) with the corresponding pixel values v_j.

  Of course we do not know the exact pixel intensity, since that is what we want to compute in the first place. However, we can try to approximate it and change the target function accordingly. Generally, we can apply any masking function $m(x)$ on the image plane and change the target function to $t'(x) := t(x)/m(x)$. In order to obtain the desired output image, we just need to multiply the result with $m(x)$. Like this we can

  - mask down directly visible light sources by setting $m(x)$ high there,

  - use a low resolution image as an approximate intensity mask, or more generally,
  - use any user-given mask.

As always, it is important to be quite conservative with the approximate guesses used to drive the sampling. If $m(x)$ is set to high for a specific area (like in the vicinity of light sources for a low resolution mask), those areas could get undersampled a lot.

2.7 Parallelization of MLT

Standard Monte Carlo based light transport algorithms are typically straightforward to parallelize, at least if we ignore that we should ensure that the seeds for the pseudorandom number generator are sufficiently independent. Usually the only sampling specific information we need in order to compute a task is a state for the random number generator or, in the case of quasi-Monte Carlo, just a single index for the sequence we use. In contrast to this the Markov chain nature of MLT, i.e. fact that the next state depends on the current state, makes it somewhat harder to parallelize things.

The obvious approach is to simulate many chains in parallel. Each thread can then simulate its own chain mostly independently of what other threads do, using a pseudorandom number generator state which is uniquely associated with that chain. In order to get reproducible results, a pool of chains can be used and iterations for all chains in that pool can be scheduled such that each chain gets the same amount of iterations.

Simulating independent chains is generally a good thing, since it reduces the amount of correlation. However, there are some things we have to keep in mind for an implementation.

Shared Framebuffer Access

Different chains, and therefore different threads, need to access the framebuffer at random positions. The requirement here is the same as for light tracing (or bidirectional path tracing in general): we need to ensure that there are no conflicts. Having a dedicated framebuffer per thread is typically not feasible if the number of threads is high, so it is preferable to use atomic instructions (if available) or streams of full samples that get transferred to the framebuffer in batches.

Start-up Bias Issues

The more chains we have, the higher the impact of start-up bias is. In addition to that, compensating start-up bias by re-sampling becomes more involved: in order to re-sample a larger number of chains, the initial set of chains to re-sample them from should to be larger accordingly. If this is not the case, there is a higher amount of correlation in the beginning, since the probability of having duplicate initial states is higher.

State Storage Size In contrast to standard (quasi-)Monte Carlo we have the requirement to store the current state. While a (quasi-)Monte Carlo estimator only needs the next sequence index or pseudo-random number generator state, MLT needs all the information to construct a mutated path based on the current path.

2.8 Summary

Metropolis light transport works particularly well in complicated scenes where other unbiased methods have trouble. There are basically two advantages it draws its strength from: First there is the powerful framework of Metropolis sampling that guarantees the right distribution and second we have the possibility to construct almost arbitrary mutations that can locally explore important illumination effects that are hard to create by other sampling methods.

- The efficiency of MLT of course massively depends on the mutation strategy that is actually implemented and whether that strategy works well for a given scene or not.
- An obvious question is: when does MLT fail? In general you can pretty much outsmart any light transport algorithm by constructing exactly the kind of scene where it falls apart. For MLT this is harder to predict though, since we need to construct a scene where all mutation strategies yield low acceptance. As such, MLT is generally very robust.

  Of course there are also many cases where Metropolis light transport does not necessarily outperform other algorithms, e.g. if the target function is sufficiently simple and traditional means of importance sampling work well enough already. For some rendering workflows this might already be the case for about every possible scene, especially if low noise for smooth areas is important right from the start and where the missing stratification hurts a lot.
- MLT poses somewhat higher constraints on a rendering system than other algorithms have.

Acknowledgments

Many thanks go to my colleagues at nvidia ARC, in particular Daniel Seibert and Dietger van Antwerpen, for many fruitful discussions and challenges not only concerning Metropolis light transport.

Bibliography

[1] Ashikhmin, M., Premoze, S., Shirley, P., and Smits, B. A variance analysis of the metropolis light transport algorithm. *Computers & Graphics 25*, 2 (2001), 287–294.

[2] Clarberg, P., and Akenine-Möller, T. Exploiting Visibility Correlation in Direct Illumination. *Comp. Graph. Forum (Proc. of EGSR 2008) 27*, 4 (2008), 1125–1136.

[3] Clarberg, P., and Akenine-Möller, T. Practical Product Importance Sampling for Direct Illumination. *Comp. Graph. Forum (Proc. of Eurographics 2008) 27*, 2 (2008), 681–690.

[4] Clarberg, P., Jarosz, W., Akenine-Möller, T., and Jensen, H. W. Wavelet Importance Sampling: Efficiently Evaluating Products of Complex Functions. *ACM Trans. Graph. 24*, 3 (2005), 1166–1175.

[5] Cline, D., Egbert, P. K., Talbot, J. F., and Cardon, D. L. Two stage importance sampling for direct lighting. In *Rendering Techniques 2006: 17th Eurographics Workshop on Rendering* (June 2006), pp. 103–114.

[6] Coleman, W. Mathematical Verification of a certain Monte Carlo Sampling Technique and Applications of the Technique to Radiation Transport Problems. *Nuclear Science and Engineering 32* (1968), 76–81.

[7] Hammersley, J. M., and Handscomb, D. C. *Monte Carlo Methods.* Wiley, New York, N.Y., 1964.

[8] Hastings, W. Monte carlo sampling methods using markov chains and their applications. *Biometrika 57* (1970), 97–109.

[9] Hesterberg, T. Weighted average importance sampling and defensive mixture distributions. *Technometrics 37*, 2 (May 1995), 185–194.

[10] Kalos, M. H., and Whitlock, P. A. *Monte Carlo Methods.* John Wiley and Sons, New York, N.Y., 1986.

[11] Kelemen, C., Szirmay-Kalos, L., Antal, G., and Csonka, F. A Simple and Robust Mutation Strategy for the Metropolis Light Transport Algorithm. *Computer Graphics Forum 21*, 3 (Sept. 2002), 531–540.

[12] Kollig, T., and Keller, A. *Monte Carlo and Quasi-Monte Carlo Methods*. Springer-Verlag, 2000, ch. Efficient Bidirectional Path Tracing by Randomized Quasi-Monte Carlo Integration, pp. 290–305.

[13] Metropolis, N., Rosenbluth, A., Rosenbluth, M., Teller, A., and Teller, E. Equations of State Calculations by Fast Computing Machine. *Journal of Chemical Physics 21* (1953), 1087–1091.

[14] Mitchell, D. P. Consequences of stratified sampling in graphics. In *Proc. of ACM SIGGRAPH 1996* (1996), Addison-Wesley, pp. 277–280.

[15] Raab, M., Seibert, D., and Keller, A. Unbiased global illumination with participating media. In *Monte Carlo and Quasi-Monte Carlo Methods* (2006), A. Keller, S. Heinrich, and H. Niederreiter, Eds., Springer.

[16] Spanier, J., and Gelbard, E. M. *Monte Carlo Principles and Neutron Transport Problems.* Addison-Wesley, New York, N.Y., 1969.

[17] Talbot, J., Cline, D., and Egbert, P. Importance resampling for global illumination. In *Rendering Techniques 2005: 16th Eurographics Workshop on Rendering* (June 2005), pp. 139–146.

[18] Veach, E. *Robust Monte Carlo Methods for Light Transport Simulation.* PhD thesis, Stanford University, 1997.

[19] Veach, E., and Guibas, L. Optimally combining sampling techniques for Monte Carlo rendering. In *Proc. of ACM SIGGRAPH 1995* (1995), Addison-Wesley, pp. 419–428.

[20] Veach, E., and Guibas, L. Metropolis Light Transport. *Computer Graphics 31* (1997), 65–76.

[21] Woodcock, E., Murphy, T., Hemmings, P., and Longworth, T. Techniques used in the gem code for monte carlo neutronics calculations in reactors and other systems of complex geometry. *Proc. Conf. Applications of Computing Methods to Reactor Problems* (1965).