A Practical Introduction to Metropolis Light Transport

David Cline*
Brigham Young University

Parris Egbert†
Brigham Young University

May 6, 2005

Abstract

Most descriptions of Metropolis Light Transport (MLT) currently in the literature focus on theoretical completeness rather than readability. In fact, the core concepts of MLT are not difficult to understand, but available descriptions of these concepts tend to be steeped in statistical terminology and light transport jargon. In this paper, we take a different approach, concentrating on giving a working understanding of the algorithm rather than theoretical concerns. It is hoped that this work will help to uncover the simplicity of MLT, and serve as a tutorial for those wishing to implement the algorithm or understand its theoretical underpinnings.

1 Introduction

One of the most interesting global illumination algorithms to emerge in the past decade is Metropolis Light Transport. First introduced by Veach and Guibas in 1997 [8], MLT uses a statistical integration technique called Metropolis Transport [5] to solve the general global illumination problem. The main strength of MLT lies in its ability to explore local portions of the space of light paths in an unbiased way. This makes MLT particularly attractive for hard sampling problems in global illumination such as caustics and light shining through small aperatures.

Since the original paper, several papers have been published that extend the basic MLT algorithm. Pauly, Kollig and Keller [6] show how Metropolis Light Transport can be used to render scenes with participating media such as smoke and fog. Kelemen et al. [3] improve the convergence characteristics of MLT by creating a novel mutation strategy. Other work that has been done, such as that by Ashikim et al. [1] seeks to describe the statistical properties of the Metropolis algorithm. In addition to these works, Pharr [7] recently published an excellent tutorial on Metropolis Sampling.

Despite the popularity of MLT as a discussion topic, relatively few implementations of it exist compared to other global illumination algorithms such as standard path

*e-mail: cline@rivit.cs.byu.edu
†e-mail: egbert@cs.byu.edu

tracing or photon mapping. Futhermore, while good practical tutorials exist on how to implement other global illumination algorithms, simple descriptions of how to implement MLT seem hard to come by. Instead, most treatments of the MLT tend to take a rigorous theoretical approach, which makes them inaccessible to those not well versed in statistical and light transport terminology. This is unfortunate, since the core concepts of MLT are not difficult to understand once the terminology in which they are cast is understood.

With this paper we attempt to strip away some of the more difficult terminology, and instead concentrate on giving a working knowledge of the concepts needed to implement MLT. Our hope is that after reading this paper, a student who has previously implemented a basic path tracer will have no trouble extending it to do bidirectional path tracing and Metropolis Light Transport. The paper should also serve as a primer for those wishing to understand the theoretical underpinnings of MLT.

The remainder of the paper will be presented as follows: we will first present the idea of Metropolis Transport in the context of image sampling, and then expand this idea to produce a version of MLT based on standard path tracing. We then give an introduction to bidirectional path tracing, which is used as the underlying sampling mechanism for MLT. Finally, we show how to link bidirectional path tracing and Metropolis sampling to produce a complete implementation of MLT.

2 Metropolis Transport

Suppose that you want to approximate a function f. One way to do this is to produce a sampling distribution proportional to f and then make a histogram of samples taken from the distribution. The resulting histogram will be proportional to f (obviously), so it only needs to be scaled to approximate f. *Metropolis Transport* uses this method to approximate functions, and can be summarized as follows:

- Create a sampling distribution proportional to f.
- Make a histogram of samples taken from the sampling distribution.
- Scale the histogram to approximate f.

In the case of an image, f is defined on some subset of $\mathbb{R}^2$, and the histogram contains one bin for each pixel in the image approximation. The scale factor, s, needed to make the histogram approximate f is the ratio of the average value of f over the sampling domain, f_{ave}, to the average number of samples per bin in the histogram, h_{ave}:

$$s = f_{ave}/h_{ave} \tag{1}$$

In practice, f_{ave} can be estimated by averaging a large number of samples selected at random from the sampling domain.

2.1 Creating a Sampling Distribution

Detailed balance. The Metropolis algorithm uses an idea called detailed balance to create a distribution proportional to f. An intuitive way to think about detailed balance is to imagine that a histogram proportional to f already exists. This distribution of samples is called the *stationary distribution*. Now imagine that a transition function exists that allows samples to flow between bins in the histogram. The stationary distribution will be maintained as long as the number of samples flowing from one bin in the histogram to another is the same as the number of samples flowing back. This property is called *detailed balance*. We will use K to denote a transition function that obeys the detailed balance condition. (See figure 1.)

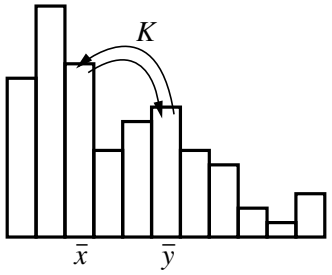


Figure 1: Detailed balance. The desired (stationary) distribution will be maintained as long as the number of samples flowing between any two bins $\bar{x}$ and $\bar{y}$ in the histogram is balanced. In other words $K(\bar{y}|\bar{x}) = K(\bar{x}|\bar{y})$.

An important consequence of detailed balance is that if a single sample is allowed to migrate in the domain of f according to K, it will produce a distribution of samples equal to the stationary distribution (proportional to the function we want to approximate). The strategy adopted by Metropolis Transport is to create a suitable K function and then use it to migrate a single sample though the domain of f. As the sample moves, a histogram is kept of its location, and this histogram is used to approximate f.

Defining the transition function. The function K is defined by using a *tentative transition function* T, also known as a *mutation strategy*. $T(\bar{y}|\bar{x})$ gives the probability of choosing point $\bar{y}$ as the proposed next sample location if $\bar{x}$ is the current sample location. To complete K, a tentative sample location $\bar{y}$ is chosen based on T and $\bar{x}$, f is evaluated at $\bar{x}$ and $\bar{y}$, and the next sample location either migrates to $\bar{y}$ with probability $a(\bar{y}|\bar{x})$, or remains at $\bar{x}$ with probability $1 - a(\bar{y}|\bar{x})$, where

$$a(\bar{y}|\bar{x}) = min\left\{1, \frac{f(\bar{y})T(\bar{x}|\bar{y})}{f(\bar{x})T(\bar{y}|\bar{x})}\right\} \tag{2}$$

The makeHistogram function in figure 2 uses Metropolis Transport to copy an image. (This is not a very useful way to copy an image, but it does provide a good example of Metropolis sampling in action.) MakeHistogram uses a very simple mutation strategy, namely choosing a random point on the image plane with a uniform probability,

```
void makeHistogram(float F[w][h], float histogram[w][h], int mutations)
{
    int i, x0, x1, y0, y1;
    float Fx, Fy, Txy, Tyx, Axy;

    // Create an initial sample point
    x0 = randomInteger(0, w-1);
    x1 = randomInteger(0, h-1);
    Fx = F[x0][x1];

    // In this example, the tentative transition function T simply chooses
    // a random pixel location, so Txy and Tyx are always equal.
    Txy = 1.0 / (w * h);
    Tyx = 1.0 / (w * h);

    // Create a histogram of values using Metropolis sampling.
    for (i=0; i < mutations; i++) {
        // choose a tentative next sample according to T.
        y0 = randomInteger(0, w-1);
        y1 = randomInteger(0, h-1);
        Fy = F[y0][y1];
        Axy = MIN(1, (Fy * Txy) / (Fx * Tyx)); // equation 2.
        if (randomReal(0.0, 1.0) < Axy) {
            x0 = y0;
            x1 = y1;
            Fx = Fy;
        }
        histogram[x0][x1] += 1;
    }
}
```

Figure 2: The **makeHistogram** function. This function uses Metropolis sampling to make a histogram from an image passed in the **F** array. It is assumed that the **histogram** array is initialized to be all zeros. After the function returns, the histogram can be scaled to approximate **F**. Below the code are several images created using the **makeHistogram** function. The original image is approximated using an average of 1 (left), 8 (middle) and 256 samples per pixel (right).

but a wide range of transition functions can be used. Later we will see that the power of MLT lies in choosing good mutation strategies.

Using detailed balance to produce the stationary distribution is very robust in the limit. It will even work if f can only be evaluated statistically (i.e. f cannot be directly evaluated but a random variable with expected value f can be). This will become an important point when the Metropolis algorithm is adapted to handle light transport.

2.2 Color Images

Up to now we have only considered how to create grayscale images using Metropolis sampling. However, the Metropolis framework can easily be extended to handle color images by redefining the histogram to accumulate in color. The luminance of the color samples at points $\bar{x}$ and $\bar{y}$ is used to define $a(\bar{y}|\bar{x})$, and colors added to the histogram are scaled to have a luminance of 1. Figure 3 shows how this might look in code. Note that any additive color space can be used for the histogram. In the common case of RGB, luminance is defined as $(0.299\ R + 0.587\ G + 0.114\ B)$.

```
DomainLocation X,Y;
.
.
for (i=0; i < mutations; i++) {
        Y = mutateAccordingToT(X);
        Tyx = T(Y, X);
        Txy = T(X, Y);
        colorY = F[Y.xloc][Y.yloc];
        Fy = colorY.luminance();
        colorY /= Fy; // scale colorY to have luminance 1
        Axy = MIN(1, (Fy * Txy) / (Fx * Tyx));
        if (randomReal(0.0, 1.0) < Axy) {
                X = Y;
                Fx = Fy;
                colorX = colorY;
        }
        histogram[X.xloc][X.yloc] += colorX;
}
```

Figure 3: Accumulating in color. Compare to figure 2. The image and histogram have both been converted to color arrays. **Fx** and **Fy** are redefined to be luminance values, and colors added to the histogram have a luminance of 1. In addition, the mutation strategy and pixel coordinates of each sample have been encapsulated. In the case of MLT, the **DomainLocation** structure will not only contain a pixel location, but will include an entire light path as well.

3 MLT Using Standard Path Tracing

Recall from section 2.1 that Metropolis Transport will work even if f can only be evaluated statistically. This is exactly the situation presented by standard path tracing. The values of f (light intensities at particular pixels in the image) cannot be computed directly. Instead, a path tracer evaluates light paths in the scene in such a way that the expected value (average) of paths contributing to a given image pixel is equal to the intensity of light reaching that pixel on the image plane. In other words, a path tracer can be thought of as a machine that uses some sampling procedure in path space to create a random variable in image space, X_f, with expected value equal to f. (i.e. $E[X_f(i,j)] = f(i,j)$) MLT is nothing more than a version of Metropolis Transport that evaluates light paths (X_f) instead of directly evaluating image pixel values (f).[1] Figure 4 gives pseudocode for an implementation of MLT.

```
for (i=0; i < mutations; i++) {
        Y = mutateAccordingToT(X);
        Tyx = T(Y, X);
        Txy = T(X, Y);
        colorY = evaluateLightPath(Y); // evaluate X_f
        Fy = colorY.luminance();
        colorY /= Fy;
        Axy = MIN(1, (Fy * Txy) / (Fx * Tyx));
        if (randomReal(0.0, 1.0) < Axy) {
                X = Y;
                colorX = colorY;
        }
        histogram[X.xloc][X.yloc] += colorX;
}
```

Figure 4: Pseudocode for Metropolis Light Transport. This procedure will work equally well for MLT defined using standard or bidirectional path tracing.

3.1 Sampling Light Paths in Path Tracing

Before we can discuss MLT mutation strategies, we must first review the sampling procedure that path tracing uses to generate light paths. By *light path*, we mean a path in ray space that connects a light source to the eye point through a number of scattering events (reflections, refractions, etc.). We will use Heckbert's regular expression notation to refer to different types of light paths [2]. In his notation, L stands for a light source, D is a non-specular surface, S is a specular surface, and E is the eye point. As

[1] The description we are using here is a departure from that used by Veach and Guibas [8]. Instead of describing light paths in terms of a random variable on the image plane, they use the path integral formulation of light transport, and describe the set of all light paths as an abstract space with high dimensionality. We believe our description to be a little more intuitive because it more closely resembles the implementations found in most path tracers.

an example, the light path LDS^*E begins at the light source, and propagates through one diffuse and zero or more specular bounces before joining with the eye point.

In a path tracer the light paths are generated by casting rays from the eye point into the scene. These *eye subpaths* (sometimes called *lens subpaths*) are then allowed to bounce around in the scene according to some probability distibution (PDF). Complete light paths are created from the eye subpaths in one of two ways. First, an eye subpath may just happen to hit a light source. We call this kind of light path an *implicit light path*. A second kind of light path is created when the end of an eye subpath is connected directly to a point on a light source. We call these *explicit light paths*. (See figures 5 and 6.)

Although a theoretically complete path tracer can be made using just implicit or just explicit light paths, typical path tracers leverage the strengths of both these path types. For example, explicit light paths are often used to compute direct lighting, whereas implicit paths are better at computing caustics and reflections of light sources.

What is being computed? When a light path is evaluated by a path tracer, it actually computes a probabilistic estimate of the light intensity (power / area) flowing to the eye point along the first leg of the light path, $L(x_1, x_0)$. This estimate, which we will call $LP(x_1, x_0)$, includes the contributions of all light sources and possible scattering events, even though only one light source and one set of scattering events is being sampled. To put it in statistical terms, the average or *expected value* of a light path is equal to the intensity of light flowing along the first leg of the path:

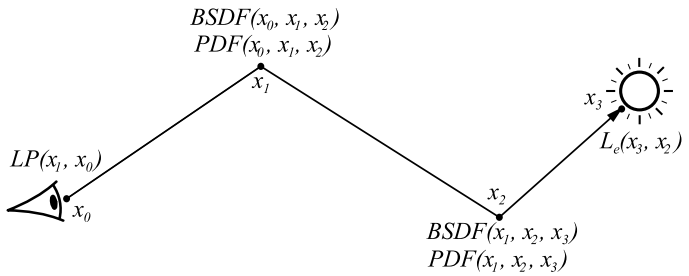
$$E[\, LP(x_1, x_0)\,] = L(x_1, x_0) \tag{3}$$

A path tracer achieves this feat by cleverly combining the light scattering properties of surfaces in the scene ($BSDFs$) and the sampling distributions used to choose directions in path space ($PDFs$). To illustrate how this is done, we will give examples for both implicit and explicit light paths.

Evaluating implicit light paths. An implicit light path is evaluated by multiplying the intensity of the light source, L_e, by the product of the bidirectional light scattering ($BSDF$) evaluated at the interior vertices of the path, and dividing by the probability (PDF) that the path was chosen by the sampling procedure.[2] Figure 5 gives an example of how this is done. Note that the path value is divided by the probability that a path of that length was created by the sampling procedure, P_{len}.

Evaluating explicit light paths. Explicit light paths are evaluated in nearly the same way as implicit paths, except the calculation starts with the power output of the light, P, instead of L_e as with an implicit light path. For a diffuse emitter, $P = A \times L_e$ where A is the surface area of the light source. To evaluate an explicit light path, the eye subpath is evaluated as in the implicit case, and a deterministic connection is made from the eye

[2] A subtle point here is that the *BSDF* and *PDF* do not actually evaluate to the bidirectional scattering or probability. Instead they give the density of these values. However, since the *BSDF* and the *PDF* are proportional to bidirectional reflectance and probability, their ratio can be used directly to evaluate the light path.



$$LP(x_1,x_0) = \frac{BSDF(x_0,x_1,x_2)}{PDF(x_0,x_1,x_2)} \times \frac{BSDF(x_1,x_2,x_3)}{PDF(x_1,x_2,x_3)} \times \frac{L_e(x_3,x_2)}{P_{len}}$$

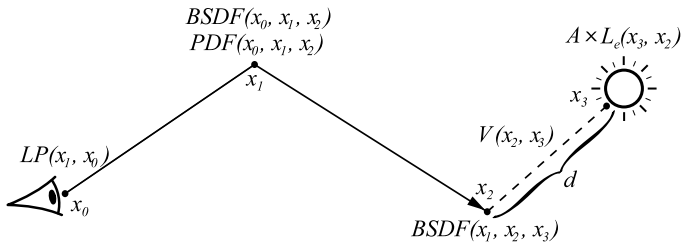
Figure 5: An *implicit light path* is created when the sampling procedure is lucky enough to hit a light source directly. The term P_{len} is the probability that the sampling procedure chose to create a path of the given length. This term is needed because $L(x_1,x_0)$ describes all the light flowing to the eye point along $x_1 \rightarrow x_0$ through any number of scattering events. Since the light path only samples one set of scattering events, its value must be "pumped up" by dividing by P_{len}.

subpath to a random point on the light source. This connection converts P to power per unit area. Figure 6 shows this graphically.

Computing the ratio of the *BSDF* and *PDF*. Often the *BSDF*s and *PDF*s in a light path can be made to cancel each other. For example, consider the case of an ideal diffuse surface. The *BSDF* scatters light in a cosine-weighted distribution about the surface normal. If directions are chosen according to a cosine-weighted *PDF* as well, the cosine weights cancel out $(BSDF/PDF)$, leaving the diffuse color of the surface. Cancelling occurs for an ideal specular surface as well. For an ideal specular surface, the *BSDF* is a delta function (it only scatters light in the reflected direction), and the only *PDF* that makes sense is another delta function centered on the reflected direction. The delta functions cancel out in the ratio, leaving the reflective color of the surface. For surfaces that are a combination of different reflection modes, such as diffuse and specular, the sampling procedure must choose which reflection mode to sample, and the *PDF* of the chosen reflection mode is multiplied by the probability that the reflection mode was chosen.

Light paths in a working path tracer. For efficiency reasons a path tracer usually makes multiple light paths from a single eye subpath. At each intersection point in the path, the sampling procedure decides whether the eye subpath will be considered an implicit light path, and whether explicit light paths should be created by connecting to light sources. A group of light paths created in this way serves as a better estimate of $L(x_1,x_0)$ than a single light path could.

Picking points on light sources. An optimization commonly used in path tracers is to modify how points are picked on light sources. For example, if points are chosen



$$LP(x_1,x_0) = \frac{BSDF(x_0,x_1,x_2)}{PDF(x_0,x_1,x_2)} \times \frac{V(x_2,x_3)\times BSDF(x_1,x_2,x_3)\times cos\theta}{\pi d^2} \times \frac{A\times L_e(x_3,x_2)}{P_{len}\times P_{light}}$$

Figure 6: An *explicit light path* is created by connecting the end of an eye subpath directly to a point on a light source. The term $V(x_2,x_3)$ gives the visibility between points x_2 and x_3, and θ is the angle between the surface normal at x_3 and the direction $x_3 \rightarrow x_2$. Besides dividing by P_{len}, the value of an explicit path must be divided by P_{light}, the probability that the particular light source was chosen by the sampling procedure. As previously stated, the division allows the light path to account for all the light reaching the eye point even though only one light source gets sampled.

at random on a spherical light source, roughly half of them will be on the back side of the sphere, and therefore occluded. One solution to this problem is to only sample points on the front side of the sphere. Another is to approximate the sphere with a disc oriented towards the intersection point. In either case, the power of the light must be scaled to reflect the new sampling strategy. In statistical terms, one random variable is simply being replaced with another that has the same expected value but a lower variance.

3.2 A First MLT Mutation Strategy

Now that we have explained how light paths are evaluated in a path tracer, we can discuss MLT mutation strategies in the context of path tracing.

Restrictions on mutation strategies. There are two main restrictions on the types of transition functions that can be used to form a mutation strategy. First, the transition probabilities $T(\bar{x}|\bar{y})$ and $T(\bar{y}|\bar{x})$ or their ratio must be computable. Second, every part of the space of light paths must be reachable from every other part. The second restriction ensures the so called "ergodicity" condition, which means that the distribution of samples will eventually converge to the stationary distribution.

Combining transition functions. It may be difficult to directly design a simgle mutation strategy that efficiently samples all of the lighting in a particular scene. It is much

easier to design transition functions around specific sampling problems, and then combine them to form a robust mutation strategy.

New path mutations. An obvious mutation strategy that fulfills both of the restrictions mentioned above is to create a new random light path at a random pixel location. We call this a *new path mutation*. For a new path mutation $T(\bar{x}|\bar{y})$ and $T(\bar{y}|\bar{x})$ are equal. New path mutations do not work well by themselves, but because they supply the ergodicity condition, they are often used as part of a complete mutation strategy.

3.3 Mutation and Light Path Density

New path mutations work because they use the same sampling procedure as a path tracer. However, many good mutations use different sampling procedures that tend to skew the density of light path samples in path space. To compensate for this effect, $T(\bar{x}|\bar{y})$ and $T(\bar{y}|\bar{x})$ must account for any changes in light path density that occur during a mutation. The following rules will allow us to derive transition probabilities for the mutations described in this paper:

- The transition probability $T(\bar{y}|\bar{x})$ is directly proportional to the probability that $\bar{y}$ is chosen as the tentative next path from $\bar{x}$, and inversely proportional to the density of paths at $\bar{x}$.
- Perturbing an angle at a diffuse surface changes the path density proportional to the cosine of the angle between the surface normal and the perturbed direction.
- Explicit changes to the pixel coordinates of a path leave the path density unchanged.
- Propagating a mutated path through specular bounces does not change the path density.
- Connecting two diffuse vertices changes the path density by an amount proportional to $|\cos\theta_1 \cos\theta_2 / d^2|$, where θ_1 and θ_2 are the angles between the surface normals at the endpoints of the connection and the connecting segment, and d is the length of the connection.
- If a connection is made in which one of the vertices is the eye point, the density change is proportional to $|\cos\theta_2 / (\cos^3\theta_1 d^2)|$, where θ_1 is the angle between the direction that the camera is facing and the connecting segment, with the other variables defined as above.

3.4 Other Mutation Strategies

Mutations starting with the eye point. Veach and Guibas describe several mutations that attempt to move the current light path starting at the eye point. The basic idea behind these mutations, which are referred to as *lens perturbations* and *multi-chain perturbations*, is to create a new light path by perturbing the pixel coordinates of the current light path. This process can be summarized as follows:

- The pixel location of the original path is perturbed a random amount in a random direction (no change in path density). See appendix 6 for details.

- Starting at the eye point, the new subpath is propagated through the same number of specular bounces as the original path. The same reflection mode is used at each path vertex as was used by the corresponding vertex in the original path (reflection or refraction). Once again, there is no change in path density.

- For a lens perturbation, the first non-specular vertex (counting from the right) is connected directly to the next vertex of the original path, which must also be non-specular, or the light source. For example, in the path *LDSSE*, the suffix *DSSE* is replaced by a new one of the same form, and the new subpath is connected directly to the point *L* from the original path. This step causes a density change proportional to $|\cos\theta_1 \cos\theta_2/d^2|$. (Note that in a path such as *LSSE*, no explicit connection is needed. The mutation simply stops when the light source is reached, and the path density does not change.)

- In a multi-chain perturbation, the outgoing direction from the first non-specular vertex (from the right) is perturbed by a random angle (density change proportional to the cosine of the outgoing angle, ϕ_i). See appendix 6 for the details of how this is done. The new subpath is then propagated through the same number of specular bounces as the original path, arriving at the next non-specular surface. If the vertex after this non-specular vertex in the path is also non-specular, the new subpath can be joined back onto the remainder of the old path. Otherwise, the path must be propagated through another chain of specular bounces. This process repeats until the old path is exhausted, or a pair of non-specular vertices is found. For example, consider the light path *LDSSDSE*. A new suffix of the form *DSE* is generated starting at the eye point. The direction of the outgoing ray from *D* is perturbed slightly and the path is propagated through two specular bounces to form the subpath *DSSDSE*. Since the next vertex in the original path is non-specular (L), the new subpath can be connected directly to the next vertex of the original path, *L*. Figure 7 shows a multi-chain perturbation graphically.

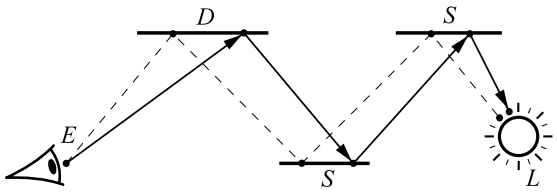


Figure 7: A *multi-chain perturbation* modifies the current path starting at the eye point. The pixel location is perturbed slightly, and the new lens subpath is propagated to the first non-specular vertex. The outgoing direction from this non-specular vertex is perturbed, and the new subpath is propagated through a specular chain looking for two successive non-specular vertices, or the light source.

There are several reasons why a lens or multi-chain perturbation may fail to create a new light path. For instance, one of the specular vertices of the original path may migrate to a non-specular surface. Also, the new path may fail to hit the light source. If either of these situations occurs, the mutation is rejected immediately. If a new light path was successfully generated, it is evaluated in the same way as the original light path (i.e. as implicit or explicit depending on the original).

Although it may be difficult to calculate the actual transition probabilities for lens and multi-chain perturbations, we can use the rules from section 3.3 to define values that are proportional to $T(\bar{x}|\bar{y})$ and $T(\bar{y}|\bar{x})$. First, it is easy to see that the probability that path $\bar{y}$ is chosen from path $\bar{x}$ is equal to the probabilitiy that path $\bar{x}$ is chosen from path $\bar{y}$. Using this fact, and the path density changes mentioned above, the transition probabilities for a lens or multi-chain perturbation are given by

$$T(\bar{x}|\bar{y}) = \left| \frac{d^2}{\cos\theta_1 \cos\theta_2} \times \prod_{1 \le i \le n} \frac{1}{\cos\phi_i} \right| \tag{4}$$

$$T(\bar{y}|\bar{x}) = \left| \frac{d^2}{\cos\theta_1 \cos\theta_2} \times \prod_{1 \le i \le n} \frac{1}{\cos\phi_i} \right|$$

Note that both transition probabilities are the same, except that $T(\bar{x}|\bar{y})$ is evaluated on path $\bar{x}$ and $T(\bar{y}|\bar{x})$ is evaluated on path $\bar{y}$. At this point, the acceptance probability $a(\bar{y}|\bar{x})$ is calculated, and based on this probability the current path either stays at $\bar{x}$, or transitions to $\bar{y}$. Finally, a sample is placed in the histogram at the current pixel coordinates.

Mutations starting at the light source. Some lighting situations can be sampled better by mutations starting at the light source instead of the eye point. Veach and Guibas describe a mutation strategy called a *caustic perturbation* that moves the current path starting at the light source. They use this mutation type to sample paths of the form LS^*DE.

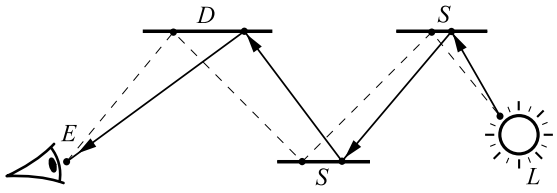


Figure 8: A *caustic perturbation* can efficiently sample paths of the form LS^*DE. The subpath LS^*D is replaced starting at the light source. The direction $L \rightarrow S$ is perturbed by a random angle, and the new direction is propagated through the same number of specular bounces as the original path. The new subpath is then connected directly to the eye point.

Caustic perturbations are created in much the same way as lens perturbations, except that they start at the light source, or second diffuse vertex in the path (from the right). The direction $L \rightarrow S$ perturbed by a random angle as in appendix 6, causing a density change proportional to $\cos\phi$. The new subpath is propagated through the same number of specular bounces as the original path, creating the subpath LS^*D or DS^*D. This subpath is then connected directly to the eye point, causing a density change proportional to $|\cos\theta_2/(\cos^3\theta_1 d^2)|$. Since the direction $D \rightarrow E$ has changed, the pixel location of the new light path has changed as well. Appendix 6 explains how to find the new pixel location. Figure 8 shows a caustic perturbation graphically.

Once again, we find $T(\bar{x}|\bar{y})$ and $T(\bar{y}|\bar{x})$ using the rules given in section 3.3. Applying these rules to a caustic perturbation yields

$$T(\bar{x}|\bar{y}) = \left| \frac{1}{\cos\phi} \times \frac{d^2 \cos^3\theta_1}{\cos\theta_2} \right| \tag{5}$$

$$T(\bar{y}|\bar{x}) = \left| \frac{1}{\cos\phi} \times \frac{d^2 \cos^3\theta_1}{\cos\theta_2} \right|$$

3.5 Estimating the Average Pixel Brightness

Recall from section 2 that once the histogram has been created, it must be scaled to approximate f. To do this, the average pixel brightness is estimated by averaging a large number of random light paths in the scene–say 10,000 or so. In our implementation, we simply take the average luminance of paths generated by new path mutations. The histogram can then be scaled to approximate the desired image by using equation 1. Figure 9 compares standard path tracing to MLT using path tracing as the sampling mechanism.

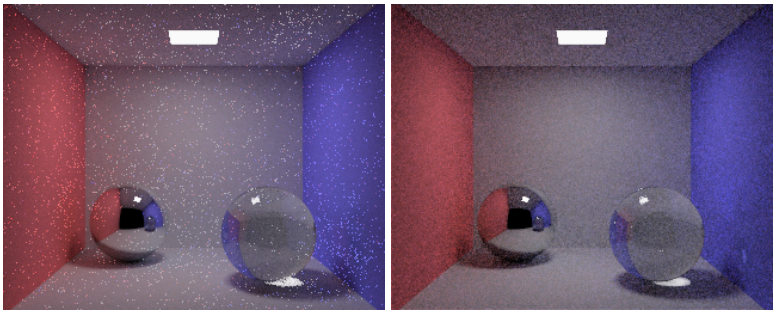


Figure 9: MLT using path tracing. The left image shows path tracing with 100 paths per pixel. The right image was rendered in approximately the same time with a path tracing version of MLT, using new path mutations, multi-chain perturbations and caustic perturbations with equal probability. As suggested by Veach and Guibas, we use standard path tracing to compute direct lighting.

4 Bidirectional Path Tracing

Bidirectional path tracing, developed nearly simultaneously by Lafortune and Willems [4] and Veach and Guibas [9] forms the basis of the Metropolis Light Transport sampling strategy. In this section, we give an introduction to bidirectional path tracing sufficient to implement MLT.

4.1 Sampling Light Paths in Bidirectional Path Tracing

Like standard path tracing, bidirectional path tracing works by sampling light paths in the scene to create a random variable with expected value equal to the desired image brightness. Unlike path tracing, bidirectional path tracing creates light paths by starting both at the eye point and the light source. As before, we will call the subpath starting at the eye point the *eye subpath*, and the subpath starting at the light source the *light subpath*.

We already discussed how to create the eye subpath in section 3.1. The light subpath is created in nearly the same way, but starting at a light source instead of the eye point. First, a random point is chosen on a random light source in the scene. A random direction also is chosen in a cosine-weighted distribution around the surface normal of the light source. This initial ray is then sent out and allowed to bounce around in the scene according to the same $PDFs$ that would be used to create an eye subpath.

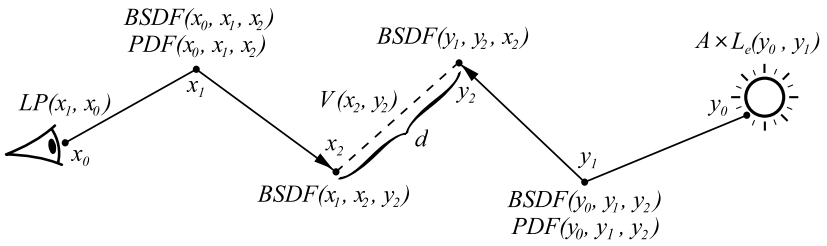
A bidirectional light path is made by connecting an eye subpath to a light subpath in a deterministic step similar to the connection made for an explicit light path. The connection is so similar that a bidirectional light path can be thought of as just an explicit light path in which the light source has been allowed to stochastically migrate through the scene. Based on this intuitive idea, it is not hard to see how to evaluate a bidirectional light path:

- Determine the power of the light source, $(P = A \times L_e)$.
- Caclulate the contribution of the eye subpath, $(BSDFs/PDFs)$.
- Calculate the contribution of the light subpath, $(BSDFs/PDFs)$.
- Connect the eye subpath and light subpath.

Figure 10 shows an example of a bidirectional light path. Note that the cosine term in the connection step has been replaced by an evaluation of the $BSDF$ function. For a diffuse surface, the $BSDF$ reduces to the cosine times the diffuse color.

4.2 Implementing a Bidirectional Path Tracer

As has just been illustrated, a bidirectional light path has many of the features posessed by an implicit or explicit light path. In particular, all three light path types have the same expected value, namely $L(x_1, x_0)$. A bidirectional path tracer is implemented in the same manner as a standard path tracer, by averaging a number of light paths at each pixel location, except that the explicit and implicit light paths used by a standard path tracer are replaced with bidirectional ones.



$$LP(x_1,x_0) = \frac{BSDF(x_0,x_1,x_2)}{PDF(x_0,x_1,x_2)} \times \frac{V(x_2,y_2)\times BSDF(x_1,x_2,y_2)\times BSDF(y_1,y_2,x_2)}{\pi d^2} \times \frac{BSDF(y_0,y_1,y_2)}{PDF(y_0,y_1,y_2)} \times \frac{A\times L_e(y_0,y_1)}{P_{len}\times P_{light}}$$

Figure 10: A *bidirectional light path* is created by connecting an eye subpath to a light subpath. The light subpath can be thought of as a stochastic migration of the light source. Compare to figure 6.

A typical implementation of a bidirectional path tracer optimizes the creation of bidirectional light paths by connecting each vertex of the eye subpath to every vertex of the light subpath. This is basically the same idea as producing many light paths from a single eye subpath in standard path tracing. When this scheme is used, care must be taken to ensure that the P_{len} terms are calculated correctly.

A problem occurs in bidirectional path tracing when the eye subpath has length zero. Since the eye point is connected directly to the light subpath, it is unclear which pixel the path should contribute to. This is the same problem that we encountered with caustic perturbations, and it can be solved in the same way, as shown in appendix 6. A second and more subtle problem is that since paths of this type are not sampled at a specified density in screen space, their value must be scaled by

$$\frac{1}{\tan(\theta_h/2)\tan(\theta_v/2)\cos^3\phi} \tag{6}$$

to convert from world space area units to screen space area units, where θ_h and θ_v are the horizontal and vertical field of view angles, and ϕ is the angle between the direction that the camera is looking and the segment connecting the eye point to the light subpath.

5 A Full Implementation of MLT

It should be fairly obvious at this point that a full implementation of MLT can be created by replacing implicit and explicit light paths with bidirectional ones in the Metropolis framework. Thus, the pseudocode in figure 4 will work equally well whether standard or bidirectional path tracing is used as the sampling mechanism. In addition, the mutation types that we have presented (new path mutations, lens perturbations and caustic

perturbations) still apply, and the changes to path density are the same whether standard path tracing or bidirectional path tracing is used. Figure 11 shows several images created using our implmentation of Metropolis Light Transport compared with bidirectional path tracing.

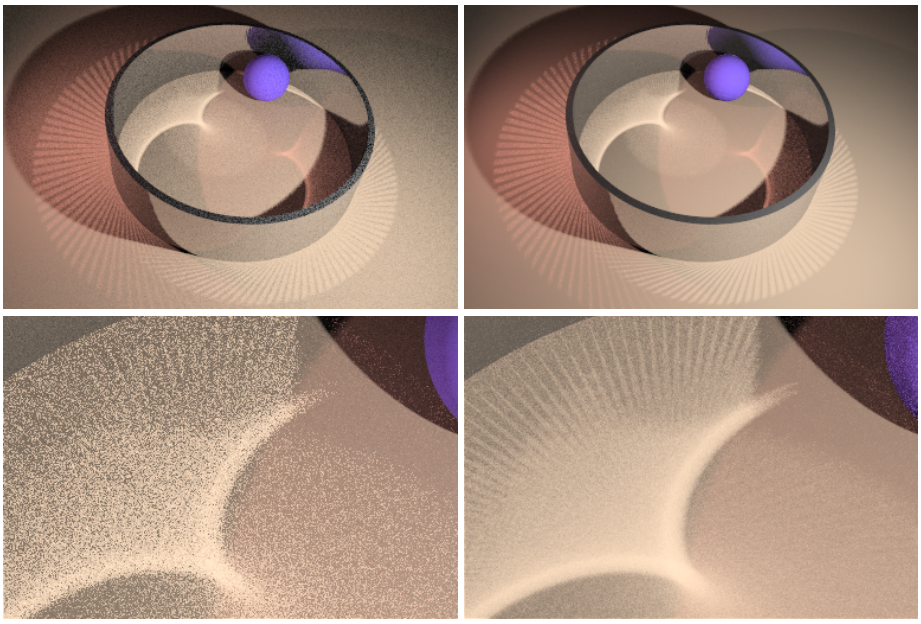


Figure 11: Comparison of bidirectional path tracing to Metropolis Light Transport. In this scene, the ring is faceted, creating detailed ray patterns in the caustics. The images on the left were made with bidirectional path tracing using 100 paths per pixel. On the right, the images were created with MLT using 100 mutations per pixel (approximately the same number of ray queries). In the top row, bidirectional path tracing achieves approximately the same quality as MLT. In the second row of images we have zoomed in on one of the caustics. This has the effect of breaking down the ability of the bidirectional path tracer to sample the caustic light paths. MLT is able to concentrate more samples on the caustic, producing a much better result. As with figure 9, we used standard path tracing to compute the direct lighting term.

5.1 More Mutation Types and Basic Optimizations

In their original paper, Veach and Guibas describe several mutation types that we have not presented, along with some basic optimizations to the MLT algorithm. The optimizations include using standard techniques such as path tracing to do direct lighting, accumulating the expected sample in the histogram, and importance sampling of mutation probabilities to increase the mutation acceptance rate. We refer the reader to [8] for the details of these techniques.

6 Conclusion

In this paper we have given an alternate formulation of Metropolis Light Transport that we believe to be functionally equivalent to the original. The new formulation relies on a statistical description of the light paths generated by standard and bidirectional path tracers rather than the path integral formulation of light transport. It is hoped that the new description will aid those wishing to implement MLT or understand the algorithm in more depth.

References

[1] Michael Ashikhmin, Simon Premože, Peter Shirley, and Brian Smits. A variance analysis of the Metropolis Light Transport algorithm. *Computers and Graphics*, 25(2):287–294, 2001.

[2] Paul S. Heckbert. Adaptive radiosity textures for bidirectional ray tracing. In *Computer Graphics (SIGGRAPH 90 Proceedings)*, volume 24, pages 145–154, August 1990.

[3] Csaba Kelemen, László Szirmay-Kalos, György Antal, and Ferenc Csonka. A simple and robust mutation strategy for the Metropolis Light Transport algorithm. *Computer Graphics Forum*, 21(3):1–10, 2002.

[4] Eric P. Lafortune and Yves D. Willems. Bi-directional Path Tracing. In H. P. Santo, editor, *Proceedings of Third International Conference on Computational Graphics and Visualization Techniques (Compugraphics '93)*, pages 145–153, Alvor, Portugal, 1993.

[5] N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller, and E. Teller. Equations of state calculations by fast computing machines. *Chemical Physics*, 21:1087–1091, 1953.

[6] Mark Pauly, Thomas Kollig, and Alexander Keller. Metropolis light transport for participating media. In B. Peroche and H. Rushmeier, editors, *Rendering Techniques 2000 (Proceedings of the Eleventh Eurographics Workshop on Rendering)*, pages 11–22, New York, NY, 2000. Springer Wien.

[7] Matt Pharr. Chapter 9: Metropolis sampling.

[8] Eric Veach and Leonidas J. Guibas. Metropolis Light Transport. *Computer Graphics*, 31(Annual Conference Series):65–76, 1997.

[9] Eric Veach and Leonidas J. Guibas. Bidirectional estimators for light transport. In *Eurographics Rendering Workshop 1994 Proceedings (Darmstadt, Germany)*, pages 147–162, June 1994.

Appendix A Projecting Points Onto the Image Plane

Given a point in world space, $P = [\, x\ y\ z\ 1\,]^T$, we would like to find its pixel coordinates in screen space. If the camera uses a lens model, a point Q is chosen on the lens, and the pixel coordinates are found by solving for the intersection point of the ray $Q \rightarrow P$ on the plane of perfect focus. On the other hand, if a pinhole camera model is used, a matrix can be derived that will project P directly onto the image plane. One such projection matrix is given as follows:

$$M = \begin{pmatrix} \frac{w}{2\tan(\theta_h/2)} & 0 & -w/2 & 0 \\ 0 & \frac{h}{2\tan(\theta_v/2)} & -h/2 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} U_x & U_y & U_z & 0 \\ V_x & V_y & V_z & 0 \\ -N_x & -N_y & -N_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & -C_x \\ 0 & 1 & 0 & -C_y \\ 0 & 0 & 1 & -C_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

where

C is the location of the camera,
N is the direction that the camera is looking,
U is the horizontal axis of the camera,
V is the vertical axis or "view up" vector of the camera,
θ_h and θ_v are the horizontal and vertical field of view angles, and
w and h are the width and height of the image in pixels.

Appendix B Calculating a Pixel Offset

The pseudocode below moves the pixel location (x, y) by an exponentially distributed random distance r between r_1 and r_2. In other words, $r = r_2 \times e^{-ln(r_2/r_1)\xi}$, where ξ is a random number between 0 and 1. This procedure is used to determine the new pixel location for a lens perturbation.

```
void pixelOffset(float r1, float r2, float &x, float &y)
{
        float phi = randomReal(0.0, 1.0) * 2 * PI;
        float r = r2 * exp( -log(r2/r1) * randomReal(0.0, 1.0) );
        x += r * cos(phi);
        y += r * sin(phi);
}
```

We use values of 0.1 pixels for r_1 and 10% of the image width for r_2. In our implementation, we do not define a *lens subpath mutation*. Instead, we use lens perturbations again, but with larger radii, 1.0 pixels for r_1 and 25% of the image width for r_2.

Appendix C Calculating an Angular Offset

The following pseudocode perturbs the direction N by a random angle that is exponentially distributed between θ_1 and θ_2. We assume that D is normalized and θ_1 and θ_2 are small. As suggested by Veach and Guibas, we use 0.0001 radians for θ_1 and 0.1 radians for θ_2.

```
void angularOffset(float theta1, float theta2, Point &N)
{
    // Make a UVN coordinate system from N
    Point U, V;
    if (ABS(N.x) < 0.5) U = N.cross(Point(1,0,0));
    else U = N.cross(Point(0,1,0));
    U.normalize();
    V = U.cross(N);

    // Determine offsets using the approximation θ ≈ sin θ
    float phi = randomReal(0.0, 1.0) * 2 * PI;
    float r = theta2 * exp( -log(theta2/theta1) * randomReal(0.0, 1.0) );

    // Calculate the new direction
    N = N + r*cos(phi)*U + r*sin(phi)*V;
    N.normalize();
}
```