Milton

Travis Fischer ‘09

Advised by Professors John Hughes and Andy van Dam

A Thesis submitted in partial fulfillment of the requirements for Honors in the Brown University Department of Computer Science

Providence, Rhode Island

Spring 2009

Contents

Chapter 1

Introduction

The goal of photorealistic rendering is to generate an image of a virtual scene that would match as closely as possible a photograph taken of the same scene in the real world. The field of photorealistic rendering has drawn the interest of many researchers with a fascination for this unique opportunity to simulate nature in its infinite complexity and has consequently flourished over the past 25 years. Today, this goal has been realized to the point where even experts are often unable to differentiate between a rendered image and a photograph, and it is this realism that has made rendering into an important tool that is used ubiquitously throughout such fields as entertainment, education, marketing, and academia. Many of these applications benefit from some theoretical guarantee of the physical correctness of the renderings they use and therefore require rendering algorithms that are physically-based in that they attempt to model the way light functions in the real world. The deeper understanding that is gained from studying the underlying physical processes may still benefit applications without these needs because more insight into these fundamentals can allow one to design better non-physically-based algorithms as well. While the original goal of photorealistic rendering has certainly been accomplished, there are still practical problems that hinder many potential students from successfully entering the field. The objective of this thesis is to address several of these problems by introducing a new rendering framework named Milton.

Milton is a cross-platform, open-source rendering framework written in C++, with an emphasis on design, efficiency, and maintaining a very high quality codebase. It is no secret that there are many rendering engines out there, but there are few, if any open source engines that fulfill all of the goals that Milton has set out to accomplish – all of the goals which are necessary for such an engine to be useful as an educational tool for others wishing to advance themselves in the field. In addition to providing a clean, yet advanced reference engine, this thesis will discuss the major software engineering and design choices embodied by Milton. We will also provide implementation details for a notoriously difficult rendering algorithm known as Metropolis Light Transport (MLT).

1.1 Thesis Organization

Chapter 2 presents a brief overview of global illumination theory, including relevant light transport definitions, the rendering equation, path integral framework, and Monte Carlo integration. **Chapter 3** relates this theory to practice, discussing several biased and unbiased rendering algorithms available in Milton. **Chapter 4** introduces Milton as a rendering framework, focusing on its major design decisions and results. **Chapter 5** concludes with a summary of this thesis' contributions and considerations for future work.

Chapter 2

Global Illumination Theory

We are mainly interested in rendering algorithms which are physically-based in that they attempt to simulate the distribution of light throughout a scene in a way which mimics how light in the physical world is distributed. Towards that end, global illumination is a term used to describe the full distribution of light throughout a scene, and in this chapter, we will give a brief overview of the theory behind global illumination algorithms in general. It is not the intent of this chapter to be a rigorous or complete introduction to the topic, but rather to give just enough of an overview and provide important definitions such that a reader unfamiliar with rendering theory will be able to follow along with the remainder of the thesis.

2.1 Assumptions

Though we would like to focus on general-purpose, physically-based global illumination algorithms in order to render as accurately as possible, we will conform to several common assumptions pertaining to the properties of light that simplify this task at a conceptual and/or computational level. Many of these simplifications result in no *perceived* difference between a render and an actual photograph for the vast majority of scenes. Since our objective is to simulate only the most relevant qualities of light necessary to satisfy the goal of photorealistic rendering, we make the following simplifying assumptions:

- We will adhere to the geometric optics model of light which looks at light from a macroscopic view and disregards the fine-grained wave and particle effects of light. This model treats photons as a continuum in which light is emitted, scattered, and absorbed only at surfaces, and travels along straight lines between these surfaces [Vea97].
- All light is assumed to be unpolarized and perfectly incoherent, and therefore, polarization, diffraction, and interference effects are ignored (many effects are then lost).
- Light between surfaces in the scene exists in a vacuum, i.e., participating media are ignored.
- The speed of light is assumed to be infinite and all measurements leading to the final render are assumed to occur instantaneously.

For many real-world environments, these assumptions prove to be insignificant, allowing us to capture the same view of a scene via rendering as would be perceived in a photograph. For this reason, almost every physically-based global illumination algorithm adheres to similar assumptions.

2.2 Light Transport

Given this simplified model of light, we will now provide the most relevant theory behind global illumination algorithms used to simulate the distribution of light in a virtual scene. We will first define a number of common terms from radiometry and tie them into the rendering equation, which concisely describes the complete flow of light throughout a scene. We will then relate the rendering equation to the equivalent, yet lesser-known, path integral framework, which more conveniently captures the structure of several popular global illumination algorithms.

2.2.1 Light Transport – Definitions

We wish to simulate the three-dimensional world that we live in, and spherical coordinates naturally arise in trying to describe the flow of light throughout that world. Related to spherical coordinates is the notion of *solid angle*, the three-dimensional analog of normal angles we are used to in 2D. Just as angles are typically measured in radians, solid angles are typically measured in dimensionless units called *steradians*, denoted by *sr*. Let $\sigma(\mathrm{D})$ denote the solid angle of set of directions $D \subset \mathcal{S}^2$ as the surface area of D projected onto the unit sphere. Solid angles conceptually measure how much of the three-dimensional field-of-view a set of directions constitute, and from this definition, it follows that there are 4π steradians in a sphere.

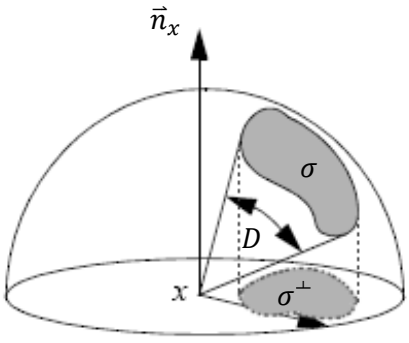

Figure 1: Solid angle $\boldsymbol{\sigma}$ and projected solid angle $\boldsymbol{\sigma^{\perp}}$ of an set of directions $\boldsymbol{D}$ on the unit hemisphere.

Another useful, related concept is a *projected solid angle*, denoted by $\sigma^{\perp}(D)$ with $D \subset \mathcal{S}^2$, which gets its name by projecting D onto the unit disc and integrating the resulting area. Projected solid angle at a surface point x with associated normal vector $\vec{n}_x$ is defined by

$$\sigma_x^{\perp}(D) = \int_D |\omega \cdot \vec{n}_x| d\sigma(\omega) \tag{1}$$

This definition implies that the projected solid angle of the entire upper hemisphere is equal to π.

2.2.2 Light Transport – Radiometry

Radiometry is the study of the physical measurement of light, and it will be helpful to be familiar with some of the physical quantities used in radiometry to measure light. Complementary to radiometry is the related field of *photometry*, which concerns itself with measuring a human observer's perception of light. Given physical measurements from radiometry, there are standardized conversions to analogous measurements in photometry, taking into account the physiology of the human eye and its response to light of varying wavelengths. For instance, a well-known radiometric property of light is that it has an associated wavelength[1], whereas the photometric equivalent would be our perception of color.

The fundamental unit in radiometry is *radiant energy*, which is often denoted by Q and expressed in *joules* (J). *Radiant power* (aka flux) is radiant energy per unit time and is denoted by Φ with units of *watts* ($W = joules/sec$).

$$\Phi = \frac{dQ}{dt} \tag{2}$$

Radiant power can be interpreted as describing the rate at which energy flows through or is absorbed by a surface per second. In relation to surfaces, *irradiance* (E) is defined with respect to a surface point x with geometric normal $\vec{n}_x$, as radiant power per unit surface area and is measured in units of $[W \cdot m^{-2}]$.

$$E(x) = \frac{d\Phi(\mathrm{x})}{\mathrm{dA(x)}} \tag{3}$$

[1] The human visual system is sensitive to electromagnetic radiation with wavelengths in the range of approximately 380 to 780 nanometers. Radiation in this visible range of the spectrum is referred to as light.

The single most important radiometric quantity for rendering is *radiance*, denoted by $L(x, \omega)$. which measures the number of conceptual photons leaving a small surface perpendicular to ω at x, per unit time, whose directions are contained in a small solid angle $d\sigma(\omega)$ around ω [Vea97]. Alternatively, using the previous definitions, radiance can be expressed as

$$L(x, \omega) = \frac{d^2\Phi(\mathrm{x}, \omega)}{dA(x)d\sigma_x^{\perp}(\omega)} \tag{4}$$

i.e., radiance is irradiance per unit projected solid angle with units of $[W \cdot m^{-2} \cdot sr^{-1}]$ (this technically doesn't account for exitant radiance because irradiance is only defined for incident directions, but this definition is commonplace and we stick to it for simplicity).

We note that all of these radiometric quantities are really wavelength-dependent, and that analogous *spectral* versions exist. It should be implied that whenever discussing radiance or one of its relatives, we really mean spectral radiance or the equivalent spectrally-dependent quantity. Spectral radiance, for example, is defined as radiance per unit wavelength, i.e., $L_\lambda = dL/d_\lambda$ with units of $[W \cdot m^{-2} \cdot sr^{-1} \cdot nm^{-1}]$.

2.2.3 Light Transport – Scattering

It will be useful to distinguish between *incident* and *exitant* radiance functions, denoted by $L_i(x, \omega_i)$ and $L_o(x, \omega_o)$ respectively[2]. Incident radiance measures the radiance arriving at a point x from direction ω_i, whereas exitant radiance measures the radiance leaving from a point x in direction ω_o [Vea97].

An important part of global illumination algorithms is modeling how light interacts with surfaces in the scene. Surfaces may be composed of many different material properties, which when looked at from a macroscopic viewpoint, may be succinctly represented by a single function, the *bidirectional scattering distribution function* (*BSDF*). The reflectance properties of a surface affect its appearance, and when light strikes a surface, though there is a lot going on under the hood, at a high level, some light may be reflected, some may be absorbed, and some may be transmitted through the surface (transparency). A mirror, for example, has very different reflectance properties as opposed to a diffuse wall. The BSDF conceptually relates incoming light to outgoing light, allowing us to represent the reflectivity and "color"

[2] We adopt the convention that ω_i always represents a vector conceptually incident on the surface and ω_o always represents a vector exitant from the surface (see Figure 2).

of a large class of materials all under one common interface. The BSDF, $f_s(\omega_i \to \omega_o)$ with $\omega_i, \omega_o \in \Omega$, is defined as the ratio of differential exitant radiance to differential irradiance and therefore has units of $[sr^{-1}]$.

$$f_s(\omega_i \to \omega_o) = \frac{dL_o(\omega_o)}{dE(\omega_i)} = \frac{dL_o(\omega_o)}{L_i(\omega_i)d\sigma^{\perp}(\omega_i)} \tag{5}$$

It will sometimes be more convenient to write the BSDF in three point form, with $x, y, z \in A$ as

$$f_s(x \to y \to z) = f_s(\frac{y-x}{\|y-x\|} \to \frac{z-y}{\|z-y\|}) \tag{6}$$

It is generally more common in computer graphics to hear about a *BRDF* (bidirectional reflectance distribution function), $f_r(w_i \to w_o)$ with $w_i, w_o \in \Omega^+$, the difference being that BSDFs are defined over the entire sphere of solid angles and therefore include transmission, whereas BRDFs are defined only on the positive hemisphere at a surface point with respect to its local geometric normal. BRDFs represent a common subset of allowable BSDFs, and they suffice in simulating the majority of real-world materials. We note that BSDFs are themselves a subclass of the more general BSSRDFs (bidirectional surface scattering reflectance distribution functions), which allow reflected light to exit from a different point on the surface as it entered, effectively allowing the simulation of subsurface scattering effects. As Milton currently doesn't support BSSRDFs, we will not touch on them further, assuming that light reflection described only at the surface is a good enough approximation for the types of commonplace materials we wish to render.

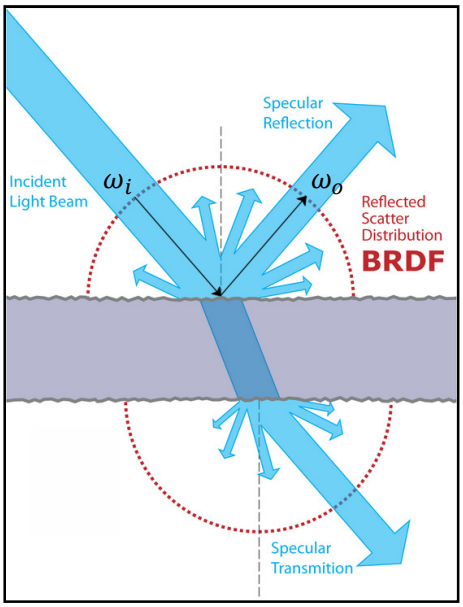

Figure 2: Geometry involved in definitions of BSDF and BRDF (image source: Wikipedia)

Physically plausible BRDFs have several key properties or constraints, namely reciprocity and conservation of energy. Reciprocity means that BRDFs are symmetric:

$$f_r(\omega \to \omega_o) = f_r(-\omega_o \to -\omega_i) \tag{7}$$

Conservation of energy states that a surface should not reflect more energy than it receives. Formally, this can be written as:

$$\int_\Omega f_r(\omega_i \to \omega_o) d\sigma^\perp(\omega_o) \leq 1 \quad \forall \omega_i \in \Omega \tag{8}$$

i.e., for any possible incident direction ω_i, the cumulative fraction of energy that is reflected over all 4π steradians must be less than or equal to one. This ensures that light propagated through a scene will converge to a steady-state distribution.

2.2.4 Light Transport – The Rendering Equation

We now have all of the background necessary to formulate an equation which will succinctly describe the steady-state radiance distribution for a given scene. Let $L_e(x, \omega_o)$ denote the radiance *emitted* from a point x in direction ω_o, and let $L_r(x, \omega_o)$ denote the indirect radiance that is *reflected* at point x in direction ω_o. Then we can break up radiance into its mutually exclusive emitted and reflected components as follows

$$L(x, \omega_o) = L_e(x, \omega_o) + L_r(x, \omega_o) \tag{9}$$

$$L_r(x, \omega_o) = \int_\Omega L(x, \omega_i) f_s(\omega_i \to \omega_o)\, d\sigma^\perp(\omega_i) \tag{10}$$

Combining these two yields the *integral formulation of the rendering equation*[3]:

$$L(x, \omega_o) = L_e(x, \omega_o) + \int_\Omega L(x, \omega_i) f_s(\omega_i \to \omega_o)\, d\sigma^\perp(\omega_i) \tag{11}$$

In order to produce an accurate snapshot of the world, we must attempt to evaluate or approximate the incident radiance at every point on the camera's film plane and for all of the possible incident directions. Much of rendering, therefore, boils down to evaluating radiance, and it is for this reason that equation (11) is so important. The rendering equation is recursive, in that the unknown

[3] Kajiya's original rendering equation was stated in a slightly different, although equivalent form.

quantity, radiance, is defined in terms of itself, thus making it particularly difficult to solve with traditional analytical methods. It is sometimes more convenient to formulate the rendering equation as a recursive integral over scene surfaces, and after performing the relevant change-of-variables, one obtains the following *area formulation of the rendering equation*:

$$L(x, \omega_o) = L_e(x, \omega_o) + \int_{\mathrm{A}} L(x, \omega_i) f_s(\omega_i \rightarrow \omega_o)\, G(x \leftrightarrow y) dA(y) \tag{12}$$

where A represents the union of all points on surfaces in the scene, $\omega_i = \frac{x-y}{\|x-y\|}$, and $G(x \leftrightarrow y)$ is known as the *geometry term* that arises from the change-of-variables as

$$G(x \leftrightarrow y) = V(x \leftrightarrow y) \frac{\left|(\vec{n}_y \cdot \omega_i)(\vec{n}_x \cdot \omega_i)\right|}{\|x - y\|^2} \tag{13}$$

$$V(x \leftrightarrow y) = \begin{cases} 1, & \text{if } x \text{ and } y \text{ are mutually visible} \\ 0, & \text{if } x \text{ and } y \text{ are not mutually visible} \end{cases}$$

Here, $V(x \leftrightarrow y)$ denotes the so-called *visibility term*, $\vec{n}_x$ is the surface normal at a point $x \in A$, and $\vec{n}_y$ is the respective surface normal at a point $y \in A$. Note that both the geometry and visibility terms are symmetric with respect to surface points and are therefore explicitly expressed as such using the notation '$x \leftrightarrow y$.'

2.2.5 Light Transport – Path Integral Framework

The light transport problem can be reformulated in a more compact, non-recursive form known as the *path integral framework*, first presented in [Vea97]. This framework recasts the rendering equation as a single integral over an abstract space of all possible paths of light, yielding a more global view of light transport which lends itself to general-purpose integration methods. In the path integral framework, an image with M pixels is created by recording a set of *measurements* $I_1, \ldots, I_M$, where I_j corresponds to the j^{th} pixel. Each pixel is assumed to lie on a virtual camera sensor which responds to incident radiance at that location in the scene through sensor responsivity, denoted by $W_e(x, \omega)$ and measured in units of $[W^{-1}]$. Each measurement can be written in the following *path integral form*[4]:

[4] See Chapter 8 of [Vea97] for more details, including a rigorous construction.

$$I_j = \int_{\Omega} f(\bar{x}) h_j(\bar{x}) d\mu(\bar{x}) \tag{14}$$

where Ω is the set of all possible paths of light, $\bar{x} \in \Omega$ is a *path* of light composed of a chain of k vertices $x_0, x_1, \ldots x_{k-1}$ with $x_i \in A$ and $k \geq 1$, $f(\bar{x})$ denotes the amount of light propagated along $\bar{x}$, $h_j(\bar{x})$ is a reconstruction filter function specific to pixel j (only dependent on the last vertex in the path), and μ is a product-area measure over Ω. This integral sums the light received by a virtual CCD sensor representing pixel j over all of the possible ways light could reach it. After evaluating an integral of this form for each pixel in the image lying on the camera's sensor, we end up with an image produced by very similar processes that would've produced a photograph of a similarly configured scene in the real world.

The main function of interest here is $f(\bar{x})$, which for our purposes resembles a function known as the *measurement contribution function*[5], and can be expressed for a path $\bar{x}$ composed of k vertices as

$$f(\bar{x}) = L_e(x_0 \rightarrow x_1) G(x_0 \leftrightarrow x_1) \left(\prod_{i=1}^{k-2} f_s(x_{i-1} \rightarrow x_i \rightarrow x_{i+1}) G(x_i \leftrightarrow x_{i+1}) \right) W_e(x_{k-2} \rightarrow x_{k-1}) \tag{15}$$

Equation (15) is obtained by expanding the area formulation of the rendering equation (12) through each of the k vertices in $\bar{x}$. It enumerates the radiance propagated along $\bar{x}$ and has a much simpler structure than the rendering equation, one in which a single expression defines the value of each measurement. By dealing with whole paths rather than rays (as is traditional when working with the rendering equation), the path integral framework provides a more explicit and complete description of light transport [Vea97]. As we will see shortly, one of the biggest advantages of the path integral framework is that it is convenient for working with probability densities on paths, which is necessary for any Monte Carlo-based rendering approach.

2.3 Sampling

The purpose of this section is to take the previous light transport equations and show how their solutions can be effectively approximated through the use of random sampling. Regardless of its form, the rendering equation's structure is too complicated to solve analytically (except for trivial cases). The best we can do in practice is to construct a random variable to approximate a solution to the rendering

[5] This definition is slightly different from that of [Vea97]; the dependence on pixel j has been separated out in accordance with the formulation of MLT, and this definition lends itself to the most intuitive implementation.

equation[6]. Furthermore, we will want to concentrate on *unbiased* estimators, whose expected value is equal to the measurement we are trying to obtain. We will give a brief overview of how Monte Carlo (MC) integration can be applied to the path integral formulation, after which we will describe several common variance reduction techniques implemented in Milton. This will provide the framework underlying nearly all of the rendering algorithms described in chapter 3.

2.3.1 Sampling – Monte Carlo Integration

Recall from equation (14) that we would like to evaluate integrals of the form

$$I_j = \int_\Omega f(\bar{x}) h_j(\bar{x}) d\mu(\bar{x})$$

Monte Carlo integration approximates this integral stochastically, sampling many random paths X_i according to some density function p defined over Ω, and using an estimator for each pixel j of the form

$$\widehat{I_j} = \frac{1}{n} \sum_{i=0}^{n} \frac{f(X_i)}{p(X_i)} h_j(X_i) \tag{16}$$

$\widehat{I_j}$ is a standard Monte Carlo estimator which exemplifies Monte Carlo integration, and it has the nice property that it remains unbiased regardless of the number of samples, n, that are taken. Observe that if each sample X_i is independently generated and if p is non-zero everywhere f is positive[7], then

$$E\left[\widehat{I_j}\right] = \frac{1}{n} \sum_{i=0}^{n} \int_\Omega \frac{f(\bar{x})}{p(\bar{x})} p(\bar{x}) h_j(\bar{x})\, d\mu(\bar{x}) = \int_\Omega f(\bar{x}) h_j(\bar{x}) d\mu(\bar{x}) = I_j \tag{17}$$

Equation (16) underlies many unbiased rendering algorithms such as path tracing, bidirectional path tracing, and their multitude of variants. In particular, these algorithms generally sample paths backwards beginning from each pixel in the output image, recording the probability density of having created any one path (either implicitly or explicitly) and the contribution of that path ($p(X_i)$ and $f(X_i)$ respectively) and evaluating each pixel estimate $\widehat{I_j}$ progressively. Metropolis Light Transport, another unbiased rendering algorithm, takes a quite different approach and will need some reworking to fit within this framework.

---

[6] Such a random variable is known as an *estimator*. An estimator for a quantity Q is a random variable, generally denoted by $\widehat{Q}$, whose expected value should be close to (ideally equal to) Q, and whose variance should be small.

[7] We have no notion of negative light, so f is assumed to be non-negative

One of the main advantages of Monte Carlo integration is its simplicity and compact structure. An unbiased estimate of this extremely complex, infinite dimensional integral is obtained after taking a single sample over path space, and a better estimate is obtained simply by taking more samples. On the other hand, one of MC integration's biggest disadvantages is its slow convergence. The standard deviation of a MC estimator reduces to the form $\sigma(\widehat{I_j}) = \frac{1}{\sqrt{n}}\sigma(X)$, which implies that in order to reduce noise in the output image by a factor of two, we must take four times as many samples, regardless of the sampling method or the dimension of the integration domain Ω.

2.3.2 Sampling – Variance Reduction Techniques

Since the main disadvantage of Monte Carlo integration is its slow convergence, many methods have been developed to reduce the variance of the X_i (which will in turn reduce the variance of $\widehat{I_j}$). Thus far, we haven't mentioned how the random paths X_i are actually generated; we will do so in chapter 3, but for now let us note several desirable properties of samplers which will generate lower variance X_i: *stratification*, *quasi MC (QMC)*, and *importance sampling*. Stratification refers to ensuring that the X_i are well distributed over the important regions of path space. QMC is a form of stratification which uses pseudo-random sequences of samples instead of random samples in order to bound the discrepancy of the resulting sample distribution, thereby ensuring stratification over the sample space.

Importance sampling attempts to draw the X_i from a probability density function proportional to $f(\bar{x})$. Note in equation (16) that if $p(\bar{x})$ were to equal $f(\bar{x})$, then our estimator would be perfect, with a variance of zero. In practice, f can be very complicated and achieving this goal is unrealistic, but by drawing more samples from the regions where f is large (concentrating our sampling effort on the "important" regions of the domain), the variance of our estimate overall is reduced, as long as we compensate for our uneven sampling rate. Think of importance sampling this way: if we have only one shot at sampling f (only one sample because of very limited resources), we would like to concentrate that sample in the region of the domain that will contribute the most to the value of the integral we are ultimately trying to approximate. By biasing our sampling technique towards 'important' regions of the domain with respect to f, we get more bang for our buck and correspondingly end up with a much lower variance estimator. If we were to instead naively sample uniformly across the domain, Ω, there's a good chance that our one, precious sample would be wasted by sampling a path, $\bar{x}$, that is unimportant, where $f(\bar{x})$ is relatively small or even zero.

Chapter 3

Global Illumination Practice

The rendering equation, path integral framework, and Monte Carlo estimator developed in the previous chapter all formulate the theory behind the global illumination problem. Many algorithms have been developed to bring this theory into practice, and in this chapter we will concentrate on several of these rendering algorithms that have been implemented in Milton, with an eye towards pointing out their assumptions, differences, advantages, and disadvantages, all within the common framework developed in chapter 2. See the results section (4.4) for a comparison of these different rendering algorithms and their implementations in Milton. In particular, we would like to focus on global illumination algorithms which have the following properties:

- *Complete* – Some global illumination algorithms require a certain representation of scene geometry or a certain description of light. We would like algorithms which share only those assumptions described at the beginning of the previous chapter. This rules out algorithms such as radiosity, though radiosity certainly still has applications in its own right.
- *Robust* – The performance of a global illumination algorithm should depend only on *what* the scene represents, rather than the details of *how* it is represented [Vea97].
- *Unbiased* – An important issue for remaining robust is bias. An unbiased algorithm is guaranteed to produce the correct result on average, whereas a biased algorithm is not. This added guarantee most often comes at the cost of increased rendering time and/or noise, but it both remains an important guarantee for certain types of applications and is also more physically-based and therefore more amenable to more realistic / general models of light transport.
- *Offline* – We limit ourselves to non-real-time algorithms, though the theory and principles discussed here are still relevant to any rendering algorithm, real-time or otherwise.

3.1 Ray Tracing

Ray tracing is a fairly naïve global illumination algorithm based on an expansion of the integral form of the rendering equation, which depending on its variant, may or may not be a full solution to the rendering equation. The rendering equation (11) may be defined with appropriate operators and expanded

as a Neumann series of the form $L = \sum_{i=0}^{\infty} T^i L_e$, where T represents a scattering operator[8]. Traditional Whitted-style ray tracing approximates this with a finite sum and generally only with a simplified scattering operator T'. An image is produced by tracing rays backwards from the eye into the scene, evaluating the direct illumination at the closest intersection point, and adding the indirect contribution by recurring with zero or more secondary reflected rays. Stochastic ray tracing describes a strategy in which variance is reduced by spawning multiple reflection rays at each intersection point, resulting in an exponential branching factor with increasing ray-tree depth (i in the previous summation).

Ray tracing is easy to implement, independent of scene representation, and captures shiny, mirror-like objects very well, though it is generally biased, not robust, and generally not a full solution to the rendering equation.

3.2 Path Tracing

Path tracing is an unbiased derivative of ray tracing that samples many paths starting through a given pixel, each of which can be seen as a random walk through path space, utilizing Russian Roulette[9] at each intersection point to either sample a reflected ray and continue the random walk or terminate. Compared to unbiased stochastic ray tracing (also utilizing Russian Roulette), which takes one or a few high quality samples which spawn many secondary rays at each intersection point, path tracing takes many low quality samples that spawn only one secondary ray at each intersection point.

Path tracing is a brute force, unbiased solution to the full rendering equation, and as such, it is the simplest and most straightforward global illumination algorithm to implement. One disadvantage of path tracing is that its performance depends a lot on the composition of the scene. Path tracing is much better at sampling certain parts of path space than others, and in particular, effects such as caustics that are formed by paths sampled with very low probability starting from the eye may take an extraordinarily long time to produce.

[8] See chapter 4 of [Vea97] for more details.

[9] Given a non-zero probability p, Russian Roulette replaces a quantity Q by an unbiased estimator $\hat{Q}$, such that $\hat{Q} = \begin{cases} Q/p, & \text{with probability } p \\ 0, & \text{with probability } 1-p \end{cases}$ such that $E[\hat{Q}] = \frac{Q}{p}p + 0(1-p) = Q$ is unbiased.

3.3 Bidirectional Path Tracing

Bidirectional path tracing (*BDPT*) is an unbiased global illumination algorithm that combines path tracing with particle tracing (path tracing starting from light sources), enabling a better distribution of samples throughout path space. BDPT was independently developed by Veach and Guibas [VG94] using the path integral framework and Lafortune and Willems [LW93] using the global reflectance distribution function (GRDF), differing only in their underlying frameworks. BDPT in Milton and in this thesis uses the path integral framework of Veach and Guibas because of its generality and subsequent use in Metropolis Light Transport (see section 3.5).

In BDPT, each measurement for $\widehat{I_j}$ independently generates a *light path* starting from a light source and an *eye path* starting from the eye, and then considers all possible ways of creating complete paths by connecting subpaths of the light path to subpaths of the eye path. This creates a family of different sampling techniques for X_i, each of which has its own probability density over path space, and each of which samples some regions of path space more efficiently than others. Samples from this family of paths are then combined using multiple importance sampling into a single estimate for $\widehat{I_j}$.

Appendix A contains the core definitions used to concisely describe and efficiently implement both BDPT and bidirectional mutations (which are based on BDPT and will be explained in section 3.5). Using the same notation as chapter 10 of [Vea97], each measurement of BDPT can be summarized as follows:

$$\widehat{I_j} = \frac{1}{n}\sum_{i=0}^{n} F_i\,, \qquad F_i = \sum_{s\geq 0, t\geq 0} w_{s,t} C^*_{s,t}\,, \qquad C^*_{s,t} = \alpha^L_s c_{s,t} \alpha^E_t \tag{18}$$

where $w_{s.t}$, α^L_s, $c_{s,t}$, and α^E_t are all defined in *Appendix A*, with $w_{s,t}$ representing the multiple importance sampling weight, α^L_s and α^E_t representing the cumulative measurement contribution weights for generating an s-length light subpath and a t-length eye subpath respectively (of the form f/p), and $c_{s,t}$ representing a deterministic connection factor for connecting the first s vertices of the light subpath with the first t vertices of the eye subpath. $C^*_{s,t}$ is known as the unweighted contribution function and is an aggregate Monte Carlo estimator of the form f/p for connecting the light subpath generated by measurement i of length s to the eye subpath generated by measurement i of length t. In this respect, equation (18) is an example of the standard Monte Carlo estimator introduced in equation (16), generating multiple paths per each sample X_i and combining them using multiple importance sampling.

BDPT uses the path integral framework to handle indirect lighting problems far more efficiently and robustly than ordinary path tracing [Vea97]. By sampling both from the light sources and from the eye, it retains the ability to sample paths that can only be sampled efficiently from one or the other (e.g., caustics are relatively hard to sample from the eye but are natural to sample from the lights, whereas direct specular reflections are relatively hard to sample starting from the lights but are natural to sample from the eye). BDPT is therefore able to converge much faster than path tracing even for relatively simple scenes. BDPT will make another appearance during the discussion of bidirectional mutations, which lie at the heart of Metropolis Light Transport, to be explained in section 3.5.

3.4 Photon Mapping

Photon mapping is an efficient, popular, biased solution to the rendering equation, developed by Henrik Wann Jensen. It is a two-pass global illumination algorithm in which photons are first traced from light sources and stored in a global kd-tree called the photon map. Final gathering is then performed by tracing paths from the eye and using the photon map to approximate irradiance at points in the scene. The first pass can be considered a special-case of particle tracing, in which photons are distributed throughout the scene from light sources, storing energy everywhere they bounce, and the second pass can be considered as a modified path tracer, augmented with lookups into the photon map and special-casing to ensure that illumination is never double-counted. For more details on the implementation of photon mapping, see Jensen's book "Realistic Image Synthesis Using Photon Mapping" (2001) and/or the open-source photon mapping implementation available in Milton.

The popularity of photon mapping has increased dramatically due to its relative simplicity and fast, smooth results. These come, however, at the cost of being biased, having a large memory overhead for storing the photon map(s), and being notorious for scene-specific parameter tweaking (e.g. number of photons, k-NN search radius, etc.).

3.5 Metropolis Light Transport

Metropolis Light Transport (MLT) is an unbiased global illumination algorithm originally introduced by Veach and Guibas in 1997. MLT is currently the most efficient light transport algorithm known for robustly rendering arbitrarily complex scenes that many other rendering algorithms would consider difficult for various reasons (e.g., strong indirect illumination, small cracks / holes, etc). At its

core, MLT is an application of the general-purpose Metropolis-Hastings sampler, applied in the context of trying to sample all possible paths of light starting at an emitter and ending at a virtual camera sensor in a carefully tuned way such that the probability density of simulating any one path is proportional to that path's relative contribution to what that virtual sensor would end up 'seeing' in the virtual scene.

The focus of this section will be to develop intuition for MLT within the Monte Carlo framework previously presented, including implementation details and intuition where possible. We will first give a brief overview of the general-purpose Metropolis-Hastings sampler and then describe how it can be applied to global illumination via MLT. This is not meant to be a rigorous formulation but rather to complement the corresponding sections of Veach's thesis and related resources in an attempt to clarify subtle points that arose during the implementation of MLT in Milton. The reader is referred to Veach's thesis for more details and to the Milton source (*src/milton/renderers/mlt*) for an example implementation.

3.5.1 Metropolis-Hastings Sampler

Given some initial state $\bar{x}_0 \in \Omega$, a sequence of random variables $X_1, \ldots, X_n$ is a called Markov chain if the probability density function of X_i is only dependent on X_{i-1} according to some transition function $p(\cdot\,|\bar{x}_{i-1})$. The chain is thought of as being "memory-less" because pending states "forget" about all previous states except for the immediately preceding state. Given some non-negative, equilibrium function $f(\bar{x})$ as a target, the goal of the Metropolis algorithm is to construct a Markov Chain such that in the limit as $n \to \infty$, the X_i are eventually distributed proportionally to f (depicted in Figure 3).

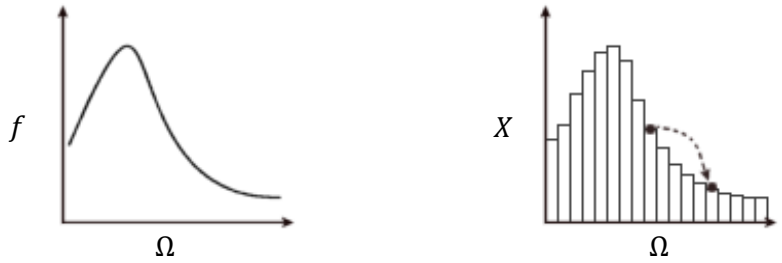

Figure 3: The equilibrium function $\boldsymbol{f}$ (*left*) and a conceptual histogram (*right*) of states visited by the Metropolis algorithm over $\boldsymbol{\Omega}$, which are distributed in proportion to $\boldsymbol{f}$. Also shown is an example of a proposed transition between two states in $\boldsymbol{\Omega}$.

The Metropolis algorithm is a random walk which carefully transitions between states X_{i-1} and X_i subject to a condition known as *detailed balance*, which says that for any two states $\bar{x}, \bar{y} \in \Omega$,

$$f(\bar{x})p(\bar{y}|\bar{x}) = f(\bar{y})p(\bar{x}|\bar{y}) \tag{19}$$

Detailed balance states that the expected flow of "energy" between any two states should be equal, where energy refers to the value of target function f. Achieving detailed balance is particularly important because the resulting Markov Chain will be distributed proportionally to f, as desired.

Let $X_{i-1} = \bar{x}$. The Metropolis-Hastings (MH) sampler achieves detailed balance by transition as follows:

- Sample a tentative state $\bar{y}$ according to a probability density function $T(\bar{y}|\bar{x})$, known as the *tentative transition function*
- Calculate an acceptance probability $a(\bar{y}|\bar{x})$ according to the Metropolis-Hastings ratio:

$$a(\bar{y}|\bar{x}) = \min\left\{1, \frac{f(\bar{y})T(\bar{x}|\bar{y})}{f(\bar{x})T(\bar{y}|\bar{x})}\right\} \tag{20}$$

- Set $X_i = \begin{cases} \bar{y}, & \text{with probability } a(\bar{y}|\bar{x}) \\ \bar{x}, & \text{otherwise} \end{cases}$

The resulting Markov Chain will converge to the desired stationary distribution if we ensure that $T(\bar{y}|\bar{x}) > 0 \;\; \forall \bar{x}, \bar{y} \in \Omega$ with $f(\bar{x}) > 0$ and $f(\bar{y}) > 0$.

3.5.2 Application to the Path Integral Framework

In order to see how this will apply to the Monte Carlo estimator $\widehat{I_j}$ that we developed previously, let us assume for a moment that we are able to draw successive paths X_i such that the probability density of sampling a path $\bar{x}$ is proportional to the luminance of that path – i.e., $p(\bar{x}) \propto f_{lum}(\bar{x})$ where $f_{lum}(\bar{x})$ represents the luminance of the light transported along the path $\bar{x}$. Then for p to be a valid probability density function, we would have:

$$p(X_i) = \frac{f_{lum}(X_i)}{b}, \qquad b = \int_{\Omega} f_{lum}(\bar{x}) d\mu(\bar{x}), \tag{21}$$

where b is a normalization constant ensuring that p integrates to one (because it is a probability density function). Substituting this new definition for p into the definition of $\widehat{I_j}$ from equation (16) yields

$$\widehat{I_j} = \frac{b}{n}\sum_{i=0}^{n} \frac{f(X_i)}{f_{lum}(X_i)} h_j(X_i) \tag{22}$$

This corresponds to taking n samples X_i, each with the same luminance (f/f_{lum} ensures that), and maintaining a histogram over the image with the end result being that pixels which receive more samples (more non-zero terms in the summation) will have a higher relative luminance than those which receive fewer samples. Because we're assuming that $p(\bar{x}) \propto f_{lum}(\bar{x})$, we expect the number of samples recorded in pixel j to be proportional to the ideal luminance of pixel j, resulting in the correct luminance histogram across the image as a whole.

This approach requires us to evaluate b and to sample from a density function proportional to f_{lum}. The Metropolis algorithm will give us the latter, and Veach explains how b can be estimated during Metropolis initialization (sections 11.3.1-11.3.2). Note in the above construction that although the constant b/n must be estimated for $\widehat{I_j}$ to remain unbiased, it is irrelevant and can safely be ignored in practice if tone mapping is applied to the output image (for most tone mapping operators which are indifferent to positive multiplicative constants when b is non-zero[10]).

Algorithm 1 Metropolis Light Transport

```
function MLT
    x̄ = x̄_0
    while patient do
        ȳ = mutate(x̄)
        α = computeAcceptanceProbability(ȳ|x̄)
        if random() < α then
            x̄ = ȳ
        end if
        recordSample(x̄)
    end while
end function
```

An overview of the MLT algorithm is given in the pseudocode above. After an initialization phase which computes the total image brightness and an initial path $\bar{x}_0$, the algorithm walks through path space by mutating the current path and either accepting or rejecting mutations according to the Metropolis-Hastings algorithm described previously. Much of MLT, therefore, concentrates on how to design mutation strategies to mutate paths such that the resulting Markov Chain will converge as quickly as possible.

[10] Milton uses so-called Reinhard tone mapping by default (Photographic Tone Reproduction for Digital Images. Reinhard et al, 2002)

3.5.3 Mutation Strategies

Although the Metropolis-Hastings algorithm guarantees that the X_i will converge correctly in the limit as $n \to \infty$, there is nothing preventing the walk from getting "stuck" for an arbitrarily long, yet finite period of time – which manifests itself in practice as extremely bright pixels caused by abnormally long chains that get stuck in a local maximum, away from which no mutation strategy could easily mutate. Path space should ideally be explored quickly and thoroughly, mixing local exploration, which is a key strength of the Metropolis algorithm, with large mutations as best as possible. Large mutations benefit from attempting to explore path space more readily at the expense of generally having a smaller acceptance ratio, resulting in long chains of the same path without successful mutation. Small mutations, on the other hand, benefit from larger acceptance ratios at the cost of either being trapped within local maxima and/or taking a long time to explore path space and thus converge. These two types of mutations directly conflict with each other, though they are both necessary for fast convergence, and it is this fundamental conflict which has gained a lot of attention recently by various researchers in the applied sciences (see, e.g. [Liu00] , [Liu01], [Cra05]).

Mutation strategies are responsible for modifying the current path in accordance with the goal of fast convergence, and in conforming to the interface presented by the Metropolis-Hastings sampler, they must be able to compute the conditional probabilities of their actions. Namely, for a mutation which proposes path $\bar{y}$ from $\bar{x}$, it must also be able to compute the MH ratio given in equation (20), comprised of $f(\bar{x})$, $f(\bar{y})$, $T(\bar{y}|\bar{x})$, and $T(\bar{x}|\bar{y})$. In accordance with this restriction, Veach proposed three main types of mutations, namely bidirectional, lens subpath, and perturbation mutations, each of which was designed with a specific purpose in mind towards facilitating fast convergence.

Bidirectional mutations lie at the heart of MLT and are responsible for handling large mutations, such as modifying a path's length. This is achieved by deleting a (possibly empty) subpath of the current path and replacing it with a (possibly empty) different subpath. In order to satisfy the ergodicity condition of the Metropolis algorithm, there is a non-zero probability that the entire path will be deleted and an entirely new subpath will be generated. *Appendix B* gives an explicit derivation for how to compute the acceptance probability $a(\bar{y}|\bar{x})$ for the case of bidirectional mutations.

Lens subpath mutations aim to stratify samples over the image plane. Each mutation consists of deleting the lens subpath of the current path and replacing it by a new one (lens subpaths are of the form (L|D)S*E). In Milton, lens subpaths are stratified across the image plane using a pseudo-random rover similar to those used in old 8-bit graphics dissolve effects. A candidate pixel for the new lens subpath is first chosen randomly and, if that pixel already has its quota of proposed lens subpath mutations, the rover

searches in a deterministic order until it finds a non-full pixel. It is worth mentioning that lens subpath mutations based only on the pseudo-random rover empirically produce strange artifacts in the output image, and it is only when combined with random pixel selection and a quota-management system that lens subpath mutations result in natural-looking, well-stratified images. Lens subpath mutations may be as simple as uniformly sampling the film plane, though the added stratification added by the pseudo-random rover yields a noticeable improvement in practice.

The idea behind *perturbation mutations* is that by slightly moving vertices along a path, you can exploit coherence while exploring nearby paths that will make similar contributions to the image, thereby increasing the acceptance probability. Perturbations are designed to complement the strengths and weaknesses of bidirectional mutations in that they make relatively small, low-cost changes to paths that have a larger acceptance probability as opposed to bidirectional mutations which are in charge of making large scale changes (making successive paths less correlated) at the expense of generally lower acceptance probabilities.

Veach proposed three types of (mutually-exclusive) perturbation mutations: lens, caustic, and multi-chain perturbations. Lens perturbations concentrate on small changes to the film plane location by modifying the lens edge of the form[11] $(L|D)DS*E$. Caustic perturbations also change the lens edge but are designed to concentrate solely on caustic suffixes of the form $(L|D)S+DE$. Multi-chain perturbations focus on paths with a suffix of the form $(L|D)DS+DS*E$. In practice, it is worth mentioning that lens and multi-chain perturbations, both of which select a new point on the film plane, can be simplified by perturbing the old film plane location uniformly with a small, square grid (e.g., 10x10 pixels) as opposed to Veach's original suggestion of using a polar perturbation with r chosen exponentially and θ chosen uniformly. The uniform solution is actually preferred because disregarding the film plane boundaries, there is no chance that the proposed mutation will fall into the same pixel as the original – which is important to mitigate the appearance of bright pixels.

3.5.4 Computing the Acceptance Probability

We will now attempt to clarify a subtlety that was not clearly addressed in Veach's PhD thesis regarding the calculation of the acceptance probability at the heart of the Metropolis-Hastings sampler. It

[11] We are using Heckbert's regular expression notation to classify paths, in which L denotes a vertex on a light source, E denotes a vertex on the eye / camera, and D and S represent diffuse and specular vertices respectively.

involves how detailed balance is affected by multiple mutation strategies and is depicted in Figure 4 below.

Figure 4: The solid lines in the figure on the left encompass all of the possible ways to transition between states $\bar{x}$ and $\bar{y}$, whereas the dashed lines in the figure on the right represent the different types of mutations which can transition between $\bar{x}$ and $\bar{y}$ (e.g. bidirectional, perturbation, etc.). A *necessary* condition for **detailed balance** is to ensure that the overall number of transitions along the solid lines are equal but a simplifying, *sufficient* condition is to ensure that transitions along corresponding dashed lines are equal.

In other words, Figure 4 states that once a mutation strategy has been chosen for sampling a candidate path $\bar{y}$, all of the other mutation strategies may safely be ignored in the computation of $a(\bar{y}|\bar{x})$ for that step of the walk. This means, for instance, that a bidirectional mutation only has to consider all possible ways of transitioning from $\bar{y}$ to $\bar{x}$ according to another bidirectional mutation (the corresponding dashed line), as opposed to considering *all* possible ways of transitioning from $\bar{y}$ to $\bar{x}$. In practice, this greatly simplifies the implementation of mutations because they only have to compute the bidirectional conditional probabilities for paths they mutated (which can be done as a byproduct during $mutate$) as opposed to calculating the transition probability density T for arbitrary paths.

Chapter 4

Milton

Milton is a high quality, open-source C++ rendering framework that is capable of advanced global illumination as demonstrated by its implementation of all of the rendering algorithms presented in the previous chapter. As an aside, we note that Milton's name comes from a mixture of MLT and the character named Milton from *Office Space*. The purpose of this chapter will be to describe the motivating principles and goals behind Milton's development, to ultimately show that these goals have been accomplished, and to present Milton as a potentially useful, open-source resource for use within the computer graphics community. After detailing these motivating principles, we will discuss the main software engineering design considerations embodied in Milton, several areas of difficulty and how our goals evolved as development progressed, and briefly touch on some results.

Milton is a moderately large software-engineering endeavor, consisting at this time of over 50k lines of well-structured C++. With that in mind, the intent here is not to cover all of Milton – for that there is separate source code documentation – but rather to focus on the high-level aspects of Milton's design which are inherent to any global illumination framework, and could, therefore, potentially be useful to other, similar engineering efforts.

4.1 Goals

Milton was designed with several, often-times competing goals in mind, and in order to understand some of the design considerations implemented in Milton, it will first be helpful to clarify the motivating principles behind these goals as follows:

- **Design** – From a computer science standpoint, what's much more difficult than the implementation of rendering algorithms themselves, is how to design and develop a modular, robust, extensible framework that is capable of both advanced, physically-based global illumination while at the same time maintaining a simple core design that is easy to understand and starting using for those with varying backgrounds in their knowledge of rendering. Creating an elegant design for such a framework is akin to how a mathematician might think about the

problem. One attempts to take the various concepts and their relationships to a given problem and ascertain their most basic, underlying principles which may then be abstracted out into their own more general, self-contained concepts and relationships. Similarly for the design of a rendering framework, maintaining a design that is sufficiently abstract with respect to rough, seemingly basic spots like surface materials, shapes, and colors so as to enable extensions, while at the same time maintaining a simple design, requires a careful balance between difficult software engineering design considerations. Arguably the most important goal during the development of Milton has been to maintain as clean and elegant of a design as possible, and section 4.2 is therefore dedicated to exploring the major design decisions put into practice by Milton.

- **Robustness** – Real world environments can be very complex in terms of geometry, materials, and lighting, and although there has been a plethora of work within the graphics community focused on emulating that complexity in a computationally feasible manner (e.g., texture mapping, various surface representations, shaders, etc.), the practical usability of a rendering engine such as Milton is bounded in a real world setting by its ability to robustly handle very complicated scenes. An important aspect of robustness, therefore, is *efficiency* and resource management. Efficiency in light of robustness has been a key goal of Milton from the start, and towards that end, Milton contains a highly optimized, concurrent ray tracing base, which remains relatively indifferent to increasing scene complexity and is practically bounded in computation time and complexity by the resources of the underlying machine (number of processor cores, memory, etc.) and not by anything algorithmic in the rendering framework itself.

- **Generality** – The majority of the rendering algorithms implemented in Milton are physically-based and unbiased. This is, however, in direct opposition to the majority of scene inputs, which although scenes in Milton are capable of retaining physical validity up to the assumptions outlined in section 2.1, the vast majority of practical scenes take advantage of some non-physically-based hack such as Phong shading or bump mapping. These hacks seem to go against some of the core principles that Milton is based upon. These techniques do, however, have their place in terms of both current utility and precedent. They are heavily engrained in both industry and academia, and for the majority of non-academic settings, bias isn't nearly as important a factor as looking "good." That being said, Milton still aims to uphold these principles, and attempts to allow for general-purpose rendering algorithms, rather than concentrating on a specific class of algorithms such as those that are unbiased. Another important part of generality

is to ensure that rendering algorithms are presented with an abstract-enough interface so as to facilitate arbitrary shape and material representations. Having a framework, for instance, which only recognizes triangular meshes, would be a limiting factor that is easily overcome with a better design. For this reason, in addition to providing our own built-in implementations of the main interfaces exposed by Milton, most public interfaces allow for dynamic behavior to be added at run-time through the use of external plugins (**.so* / **.dll*).

- **MLT** – Several design decisions centered around one of our most prevalent goals, which was to support Metropolis Light Transport.

4.2 Design

With these goals in mind, we will now delve into areas we came across during the design and development of Milton which were either notably interesting and/or challenging. We will describe the material representation in Milton, the scenefile format and related *PropertyMap* paradigms, the core *Path* data structure, several high level optimizations implemented in Milton, the *PointSampleRenderer* design, and conclude with a discussion of several problems of the current design.

4.2.1 Material Representation

Materials in Milton aim to be physically-based in the sense that a point on the surface of a primitive is characterized by its reflective, emissive, and sensor response properties. Reflectance is defined by a BSDF hierarchy, and emitters / sensors are defined similarly. Milton does not currently support BSSRDFs which would allow for the simulation of subsurface scattering. BSDFs built into Milton include lambertian, thin dielectric (including Fresnel's Laws), energy-conserving Phong (see Figure 5), absorbent, and an aggregate hierarchal BSDF composed of one or more sub-BSDFs. All built-in BSDFs allow for efficient importance sampling, and it worth noting that sampling from a BSDF uses the same *Sampler* interface that other, more standard, samplers use such as the built-in normal sampler which draws samples $X_i \; IID \sim N(\mu, \sigma)$. This common sampling interface allowed us, for example, to implement generic multiple importance sampling, where the only difference between different *Sampler* implementations is the type of variant returned by the *sample* method.

Ideally, one would have the ability to define an arbitrary mapping between surface points (or for that matter, any point — participating media / subsurface effects, etc.) and an associated BSDF defined at a given point. A *Material* would then, from a design standpoint, be a function which takes in a surface point (generally specified by *u, v* coordinates across the surface) to a specific BSDF defined at that point. This is a very generic and physically-based solution to the design of a material system for a rendering engine such as Milton, but we have chosen to go with a slightly simpler option. In Milton, instead of having a completely arbitrary mapping between surface points and their associated BSDFs, each surface and its associated material define a mapping to a single parameterized BSDF, whose **parameters** are allowed to vary with respect to the given surface point. This is useful for a variety of empirically or physically-based BSDF models with user-defined inputs that allow a single BSDF model to capture the appearance of a wide range of real-world materials. BSDF input parameters may be constant along the surface of the material or vary according to filtered lookup in an image defined over the (*u*, *v*) coordinates of the surface. This same parameterization holds for emittance defined across a surface as well as so-called sensor response, which indicates how responsive a point is measuring incident radiance (sensor responsivity is non-zero only for points on the surface of a camera).

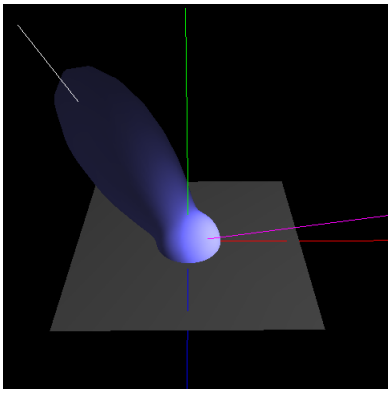

Figure 5: Modified Phong BSDF Visualization (reciprocal and energy-conserving)

The lambertian BRDF, for instance, models a perfectly diffuse surface whose exitant distribution of energy is uniform over the hemisphere without respect to the incident angle of light. It takes as input a single parameter per sampled wavelength, referred to as k_d, representing the fraction $[0, 1]$ of reflected versus absorbed spectral radiance at a given wavelength. This parameter, k_d, is allowed to vary along the surface of the *Material* as input to the single *DiffuseBSDF*, via lookup in an image defined over the (*u, v*) coordinates of the surface. This is commonly known as a texture map, though it is fundamentally more general in that it can be used to parameterize BSDFs in many more ways than just modulating the surface "color" with position.

Materials are defined without respect to the underlying surface such that a single *Material* may be referenced from many different shapes. Each distinct *Shape* instance in the scene has a reference to its associated *Material*, and upon intersection, a *Shape* queries its *Material* to lazily fill in an associated *BSDF*, *Emitter*, and *Sensor* for the intersected *SurfacePoint*. A *SurfacePoint* is a core data structure which encapsulates all of the relevant geometric information about a point on the surface of a *Shape*, in addition to providing everything necessary for shading evaluation (*BSDF*), emittance evaluation (*Emitter*), and sensor response evaluation (*Sensor*). *SurfacePoints* explicitly support the useful distinction between geometric and shading normals defined at a point, where the geometric normal is the actual normal of the underlying surface, and the shading normal is a (possibly) perturbed version of the geometric normal. Normal vectors are primarily used in dot products, and in order to cleanly support shapes such as point lights or pinhole cameras which don't really support the notion of a normal vector, all normal dot product requests go through a normal vector proxy contained in the *SurfacePoint*, which takes in a *Vector3* and returns one if the underlying surface does not support normals, or the actual dot product of the shading normal with the given vector otherwise. This decision has allowed us to naturally handle point lights (which arguably have their uses) without the need for special-casing.

4.2.2 Scenefile Format

Many classes in Milton are customizable via a number of input parameters, and it would both be awkward and decrease readability to have those classes take in all possible combinations of inputs as parameters either via constructors or accessors / mutators. We chose to instead parameterize these classes at run-time via a common base class called a *PropertyMap*, which represents a key-value mapping between *std::string* keys and a set of well-defined primitive variant-value types (*int*, *double*, *string*, etc.). By taking advantage of *boost::any*, a template variant holder utility class provided by the boost library, all code dealing with parameters became both more extensible and more readable at the expense of static type-checking in several places. This allows parameters for most core Milton classes to be specified via the commandline or more commonly in the scenefiles themselves, and loose type-checking is instead done at run-time during scenefile parsing. This setup allows for a lot of flexibility, both within Milton and in terms of allowing external extensions. One could, for example, use the same built-in loading code and scenefile format but add his/her own BSDF type with its own arbitrary key-value inputs, as long as the value-types in question are supported variants, without having to change anything within the Milton library itself. Milton also provides a global *PropertyMap* encapsulated in a *ResourceManager* which stores global program options (specified, for instance, on the commandline), as well as a global thread-

local *PropertyMap* allowing for synchronized, thread-specific storage. The global thread-local storage is used, for example, in Milton's implementation of Jensen's *PhotonMap* in order to store a single, preallocated photon buffer per render thread.

Closely related to the loose nature of *PropertyMaps* is the Milton scene file format. In order to fulfill our goals of having scenefiles that were going to be both concise, well-defined, and highly extensible via arbitrary key-value pairs, we ended up defining an implementation-independent context free grammar to capture the structure and types that would be used in Milton scenefiles as well as a concrete implementation of that grammar via the popular JSON[12] format. Figure 6 depicts an example Milton scenefile using the JSON file format containing a single area light source and rendered with the built-in path tracer.

```
{
   "scenefile" : {
      "renderer" : {
         "type" : "pathTracer",
         "noRenderThreads" : 10,
         "noSuperSamples" : 1000
      },

      "camera" : {
         "type" : "thinlens",
         "eye" : [ 0, 10, 0 ],

         "aperture" : 50,
         "fstop" : 18
      },

      "scene" : {
         "material" : {
            "emitter" : {
               "type" : "oriented",
               "power" : [ 80, 80, 80 ]
            },

            "plane" : { }
         }
      }
   }
}
```

Figure 6: Typical JSON scenefile example. Keywords in blue are aggregate types which accept child elements.

[12] JSON stands for Javascript Object Notation and is a valid subset of the Javascript language used commonly in webapps for data exchange due to its concise, human-readable format

Many aspects of the scenefile, including the camera, output, and renderer are all optional, and in the case that they aren't provided, reasonable defaults will be used. What's great about JSON is that its own design (and Javascript in general) was guided by the recursive definition of objects as unordered lists of key-value pairs similar in spirit to our extensible *PropertyMap* paradigm. For more information on the Milton scenefile format, in-depth documentation is available at *http://milton.mjacobs.net/docs/scenefile/*.

4.2.3 Path Data Structure

Paths of light are the central unit in the path integral formulation of light transport, upon which path tracing, bidirectional path tracing, and MLT are all founded (Milton's implementation of photon mapping also utilizes the same path structure for convenience). The data structure used to store and manipulate paths is therefore a very important part of any global illumination algorithm. In Milton, a *Path*, that we'll denote by $\bar{x}$, is composed of k *PathVertex* vertices, $x_0 x_1 \dots x_{k-1}$ with $k \geq 0$. Each *PathVertex* stores reference-counted *SurfacePoint* (which includes relevant associated metadata, UV cords, normal vector, BSDF, etc.), as well as a cache of carefully chosen information local to that *PathVertex* in its parent *Path*. For a *PathVertex* x_i, this extra information includes:

$$G(x_{i-1} \leftrightarrow x_i),\ G(x_i \leftrightarrow x_{i+1}),\ \alpha_i^L,\ \alpha_i^E,\ p_i^L,\ p_i^E,\ f_s(x_{i-1}, x_i, x_{i+1}),\ \overrightarrow{p_{\sigma^\perp}}(x_i|x_{i-1}),\ \overleftarrow{p_{\sigma^\perp}}(x_i|x_{i+1}) \tag{23}$$

Of this data, α_i^L, α_i^E, p_i^L, and p_i^E all store cumulative information about either the light or eye subpath of the parent path up through and including vertex x_i (see *Appendix A* for a summary of these definitions or Veach's thesis for detailed explanations). A *PathVertex*'s BSDF can be queried for whether or not that vertex represents a symbolic Dirac delta distribution commonly referred to as a *specular* vertex. These vertices require and/or benefit from special handling in several global illumination algorithms.

The *Path* data structure contains methods for *appending* a vertex onto a light subpath by either sampling the BSDF of the last vertex, or, in the case that the path is currently empty, randomly sampling an initial vertex from the light sources in the scene. *Path* contains a similar method for *prepending* a *PathVertex* onto an existing eye subpath by either sampling the adjoint BSDF of the first vertex in the path, or, in the case where the current path is empty, sampling a location on the film plane. All mutable *Path* operations record the corresponding probability densities of their actions for later use (and an *append* or *prepend* operation may optionally use Russian Roulette, in which case the compensating factor is implicitly rolled into the path's cached probability density). *Paths* may also be appended onto each

other or broken into subpaths, all the while retaining both valid local and cumulative information in every *PathVertex* relating to the current *Path* as a whole. Implicit reference-counting is utilized for dynamic *PathVertex* data because of the frequency of copying that goes on during *Path* manipulation and because one *SurfacePoint* can be referenced by multiple *PathVertex* vertices from different *Paths*. In addition to manipulation operations, *Path* contains methods for efficiently querying the contribution and cumulative probability densities of any valid subpath. This functionality is necessary for bidirectional path tracing and also simplifies the code for computing the acceptance probabilities of bidirectional mutations in MLT.

4.2.4 Optimization

Efficiency is a very important prerequisite for robustness, and Milton contains several features with the goal of efficient rendering in mind. Milton is highly multithreaded, and the core rendering speed is practically limited by the number of cores on the underlying machine. The number of rendering threads is customizable at run-time via scenefiles, and multithreading is cross-platform thanks to Qt. We have also implemented an efficient, templated linear algebra library that's accelerated through the use of *SSE* (Streaming SIMD Extensions) which, for example, allow two 4-element vectors to be added together in a single assembly instruction using specialized 128-bit *SSE* registers.

The main computationally complex operation inherent to most ray-based rendering algorithms is computing intersections between rays (vectors in $\mathbb{R}^3$ coupled with an origin point) and a collection of objects constituting the scene. We are very proud to say that at the heart of Milton sits an extremely hand-optimized kd-Tree, which may be constructed via one of several built-in heuristics (the default of which is the Surface Area Heuristic, *SAH*). A great deal of time was spent implementing an as-efficient-as-possible (without resorting to assembly) spatial acceleration scheme, and the result is that Milton renderers are capable of handling large scenes with billions of primitives. In practice, rendering speed grows sub-log-linearly with respect to the number of primitives due to the nature of the SAH and the optimized kd-Tree traversal routine which incorporates aggressive culling for cases where early termination is possible. kd-Tree build times are, however, noticeable for larger scenes due to the $O(n \log n)$ complexity of the SAH version that we've implemented. Note that due to the scenegraph-like format of Milton scenefiles, multiple levels of recursive kd-Trees are likely to be created per-scene (meshes, for example, always contain their own sub kd-Tree). This has the advantage of making it unlikely for a single kd-Tree to become unwieldly in size and approach the limits of memory on the underlying platform. It does, however, have the disadvantage that the weighting scheme for top-level kd-

Trees used in the surface area heuristic may be degraded because sub-primitives may no longer be approximately equal in terms of expected traversal cost (i.e., traversing a node containing a large, child kd-Tree will be much more expensive than traversing a node containing a child primitive, but this knowledge is not currently utilized anywhere). A scheme to balance the SAH based on more knowledge of the recursive kd-Tree structure would be an interesting topic for further research.

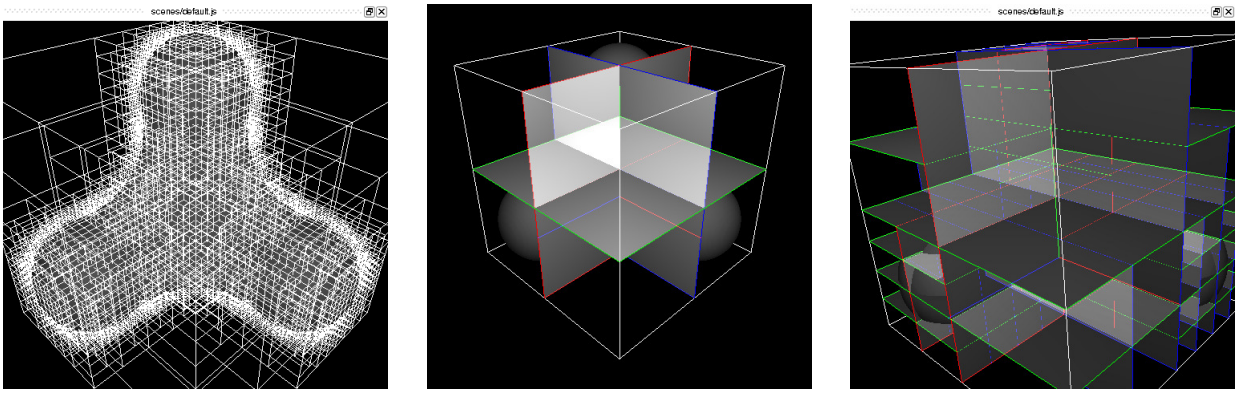

Figure 7: (*left*) OpenGL visualization of a kd-Tree built with a spatially-uniform splitting strategy, yielding a data structure that is functionally equivalent to an octree. (*middle)* Alternative visualization utilizing transparency of the same octree after three splits. (*right*) A third OpenGL visualization of a more adaptive kd-Tree built with the SAH. Note that all primitives, materials, and lights in Milton have the ability to *preview* themselves in OpenGL for debugging and development purposes. This has also made Milton useful in other, non-rendering computer graphics applications as a general-purpose visualization tool.

4.2.4 PointSampleRenderer

Besides MLT, every rendering algorithm that we've discussed attempts to evaluate incident flux over the camera lens by taking many point samples along the film plane, typically in proportion to the number of pixels in the desired output image, and then reconstructing an image from those samples by convolving the point samples with some type of reconstruction filter approximating the ideal sinc[13].

There are many possible ways of selecting sample points on the film plane which might make sense, and the selection of sample points is inherently independent of their subsequent evaluation (with the exception of adaptive sampling). By completely separating the process of generating point samples from the relatively expensive process of evaluation, we can easily support an arbitrarily large number of core render threads, whose purpose is to evaluate point samples via whatever rendering algorithm is

[13] Reconstruction filters available in Milton include uniform, linear, Gaussian, Mitchell, and Lanczos sinc.

currently selected, without changing or complicating the sample generation code at all. All Milton renderers besides the OpenGL preview renderer are subclasses of *PointSampleRenderer*, which acts as a shared buffer in a standard producer-consumer problem. *SampleGenerator* threads produce sample points over NDC in $[0,1]^2$ (normalized device coordinates representing the camera's viewport), and a customizable number of render threads consume those samples by evaluating them and adding the resulting incident radiance values to a synchronized *RenderOutput*. The *RenderOutput* is abstract and may be either piping samples out to a file, an image generated with a reconstruction filter, a canvas in the Qt GUI which comes bundled with Milton, or possibly a user-defined plugin.

4.2.5 Problems with Current Design

Several primitive shapes are currently unable to calculate their world-space surface area correctly after undergoing arbitrary non-affine transforms. Another problem that's more software engineering-oriented in nature is that while the *PropertyMap* design pattern is very extensible and clean to use, it does so by sacrificing strict compile-time type checking and is thus susceptible to type mismatches at run-time (via *boost::bad_any_cast* exceptions). This possibility is currently minimized by excessive type checking during scenefile parsing to ensure that dynamic parameters match their expected types, at least for built-in classes. While this does not provide the type of strong guarantee one would like from a typing system, we have found it to be adequate during testing to catch accidental type errors in scenefiles. Another problem related to the poor guarantees of *PropertyMaps* is that there is currently no standard way for classes that utilize and expect dynamic parameters from PropertyMaps to 'publish' this information in a way that could be used by the scene loader and/or a better auto-documentation scheme. Expected parameters for classes exposed in scenefiles are currently not tied to the class' implementation, which will inevitably lead to inconsistencies in documentation over time (source code documentation, however, is auto-generated using *Doxygen* to resolve this problem).

From a physical point-of-view, there are places where Milton does a poor job of ensuring that units match up properly and that every calculation has a corresponding physical interpretation. The goal of approaching rendering from a physically realistic angle often lost out to the somewhat conflicting goals of efficiency and readability. We encourage other rendering engines to consider early on in their design the degree to which they will ensure the accuracy and consistency of physical units.

A common design pattern for classes within Milton is to have an *init* method in addition to the constructor, whose purpose is to initialize the object after the caller sets up any possible metadata that

may be necessary for it to do so. While this design pattern has the advantage of allowing for more flexible, lazy initialization, in practice, it has proven unintuitive for other programmers to get used to, and a redesign of the *init*-before-use paradigm might be worth considering.

The scenefile format in Milton includes optional *renderer*, *camera*, and *output* definitions, in addition to the required *scene* definition. If we were to redesign the high-level scene format, we would instead opt to separate out the main content (the *scene* and *camera* definitions) from its presentation (*renderer* and *output* fields), by placing the latter in its own *milton.ini* file to be loaded separately upon running Milton. Allowing individual scenefiles to override defaults for renderer-specific parameter does, however, have its advantages; photon mapping, for instance, is notorious for its dependence on scene-specific parameters.

4.3 Difficulties

One particularly annoying difficulty in any rendering engine such as Milton is the correct handling of so-called specular surfaces. Specular surfaces simulate perfect mirrors by having a BSDF which is represented by a Dirac delta distribution (which is non-zero over one or a finite set of directions with measure zero), making them particularly difficult to simulate under a traditional Monte Carlo framework since they are impossible to sample successfully with probability one. Specular surfaces are much more efficiently handled by deterministic sampling in which the probability density function is also a Dirac delta distribution akin to sampling a discrete, cumulative distribution function. The infinite densities and corresponding infinite BSDF values that arise during simulation are difficult to work with and combine with cumulative MC estimates along the path as a whole. And though perfect specular surfaces don't technically exist in the real world, they have nevertheless been engrained as a standard part of computer graphics, originally due to the fact that their deterministic sampling makes them easy to simulate in non-MC frameworks. Furthermore, if we were to disregard perfect specular reflectors, we would still have to work with highly-peaked BSDFs in order to represent polished glass – an alternative which comes with its own pitfalls, given that floating point numbers lose precision fast when working with extreme-valued functions (both large and small).

In order to simulate specular surfaces, most rendering frameworks separate diffuse reflection out from specular reflection (where diffuse reflection refers to non-perfectly-specular reflection), and in this respect, Milton is no different. BSDFs in Milton have a flag denoting whether they represent a symbolic Dirac delta distribution or a normal BSDF, coupled with a finite probability density function. This

solution is somewhat unsatisfying as it necessitates a great deal of special-casing in the rendering algorithms themselves. One of motivating principles behind any rendering Framework like Milton is to try and abstract out all of the difficult and/or unpleasant commonalities that rendering algorithms may run into, and in this respect, we have failed to do so for specular surfaces. And while there is certainly something to be said for having the ability to simulate specular surfaces differently if a rendering algorithm wishes to do so (photon mapping is a great example of this approach), there is certainly also something to be said for keeping the exposed API (of BSDFs, for example) as simple as possible and not forcing external algorithms to adhere to special-casing that is not intrinsic the physics which are being simulated. Along these lines of thought, one idea we had during development which never got put into Milton, would be to provide wrappers for both the representations of BSDF values and probability densities which would use operator overloading to maintain internal, symbolic representations that implicitly adhered to strict rules for common mathematical operations such as multiplication and addition. These wrappers would symbolically ensure the correctness of any computation involving Dirac delta distributions and would provide for easy, implicit conversions between discrete CDFs and continuous PDFs. BSDFs could still expose the specular flag, but at least this way wouldn't necessitate an extra complication of rendering algorithms and would allow for some algorithms such as bidirectional path tracing and MLT to be implemented much more compactly and elegantly by removing all special-casing. Again, this approach was never implemented into Milton due to time constraints, but it seemed interesting enough to mention.

Another practical difficulty we ran into with MC-based rendering algorithms was a singularity inherent to the geometry term[14], $G(x \leftrightarrow y)$, used to convert between solid angle and area measures. G is unbounded due to the division by distance squared, which in practice, amounts to singularities in corners and near edges of objects. Debug builds of Milton attempt to check for these singularities through the religious use of assertions to ensure that possible affected values do not result in NaNs or infs, though the unbounded nature of the geometry term can still lead to poor simulation of paths which contain adjacent vertices that are arbitrarily close to each other.

[14] See equation (13).

4.4 Results

In this section, we provide a brief comparison of results obtained using the different rendering algorithms available in Milton. We begin with a *Cornell Box*[15] scene containing a transparent dragon mesh comprised of around 40k triangles.

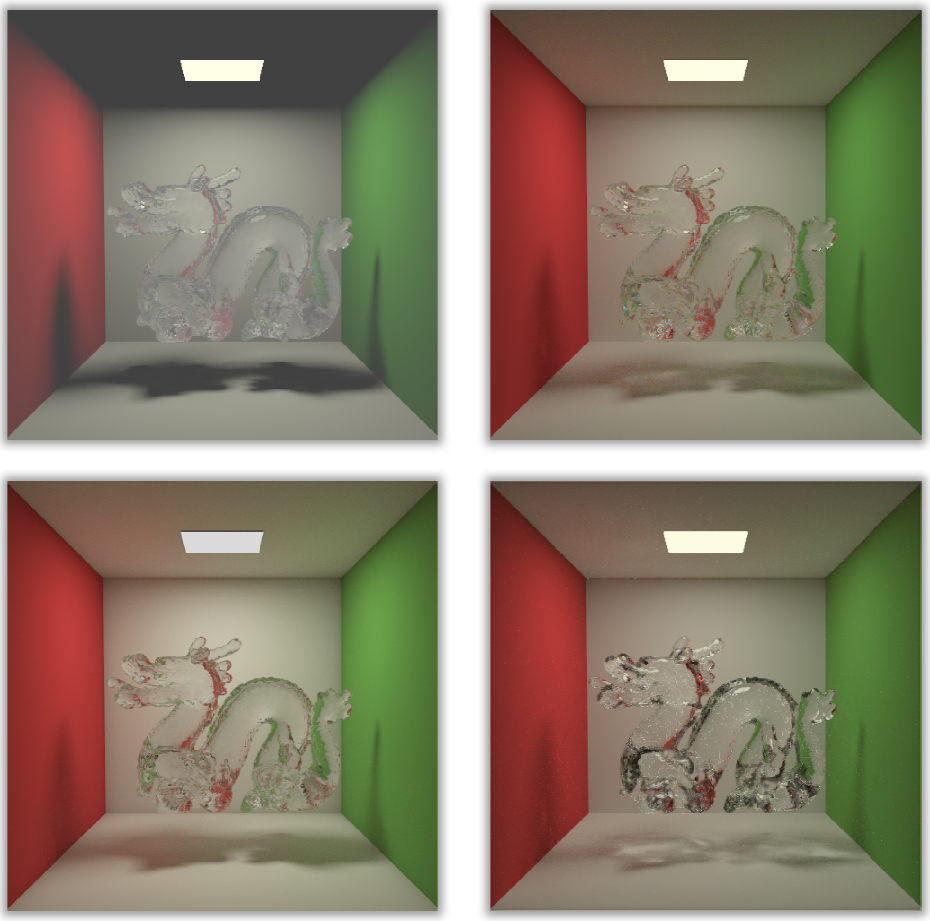

Figure 8: The same scene rendered using *ray tracing* (top left), *path tracing* (top right), *bidirectional path tracing* (bottom left), and *MLT* (bottom right). The brute force path traced version in the top right should be seen as a correct, reference image. Note the lack of caustics on the floor and lack of indirect illumination in the ray traced version. Discrepancies in the bidirectional path tracing and MLT renders are due to differences in tone mapping as well as implementation issues with correctly handling specular paths.

[15] The Cornell Box scene was developed by Cornell University as a standard test scene for physically based rendering.

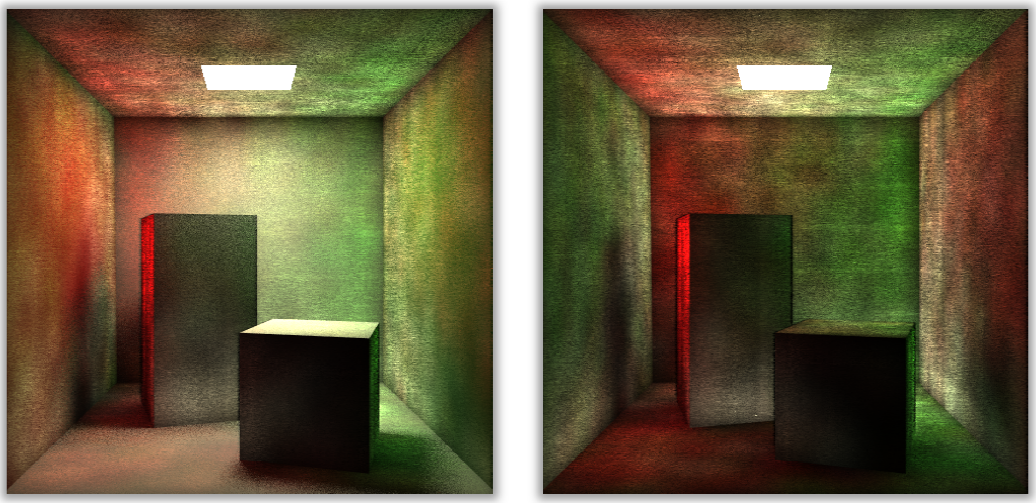

Figure 9: Two interesting bloopers of the Cornell Box which occurred while debugging photon mapping as a result of unsynchronized access from multiple render threads to the global, diffuse photon map.

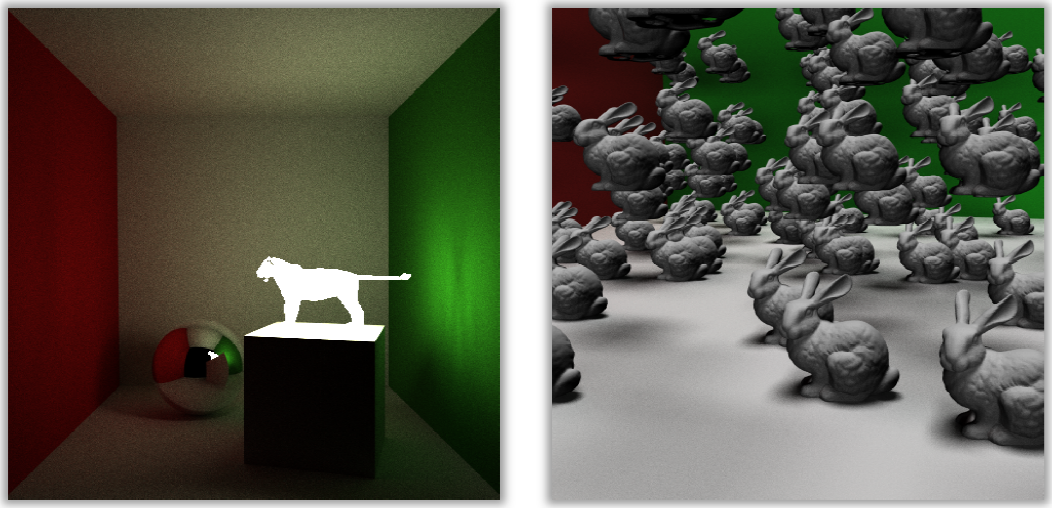

Figure 10: (*left*) All primitive shapes in Milton allow for samples to be taken across their surface and evaluation of world-space surface area. Any shape, including meshes, may therefore be used as an emitter, providing for some cool effects such as the lion emitter path traced in the image on the left. The image on the *right* contains 1000 high resolution (scanned) Stanford bunny mesh *instances*, each of which contain about 70k triangles yielding around 70 million triangles overall. The scene uses two SAH kd-Trees, one for the master bunny model (shared by all instances), and one for the scene overall, allowing this complicated scene to render via path tracing at 16 samples per pixel in approximately 5 minutes (on a quad-core machine with 8 render threads and two area light sources).

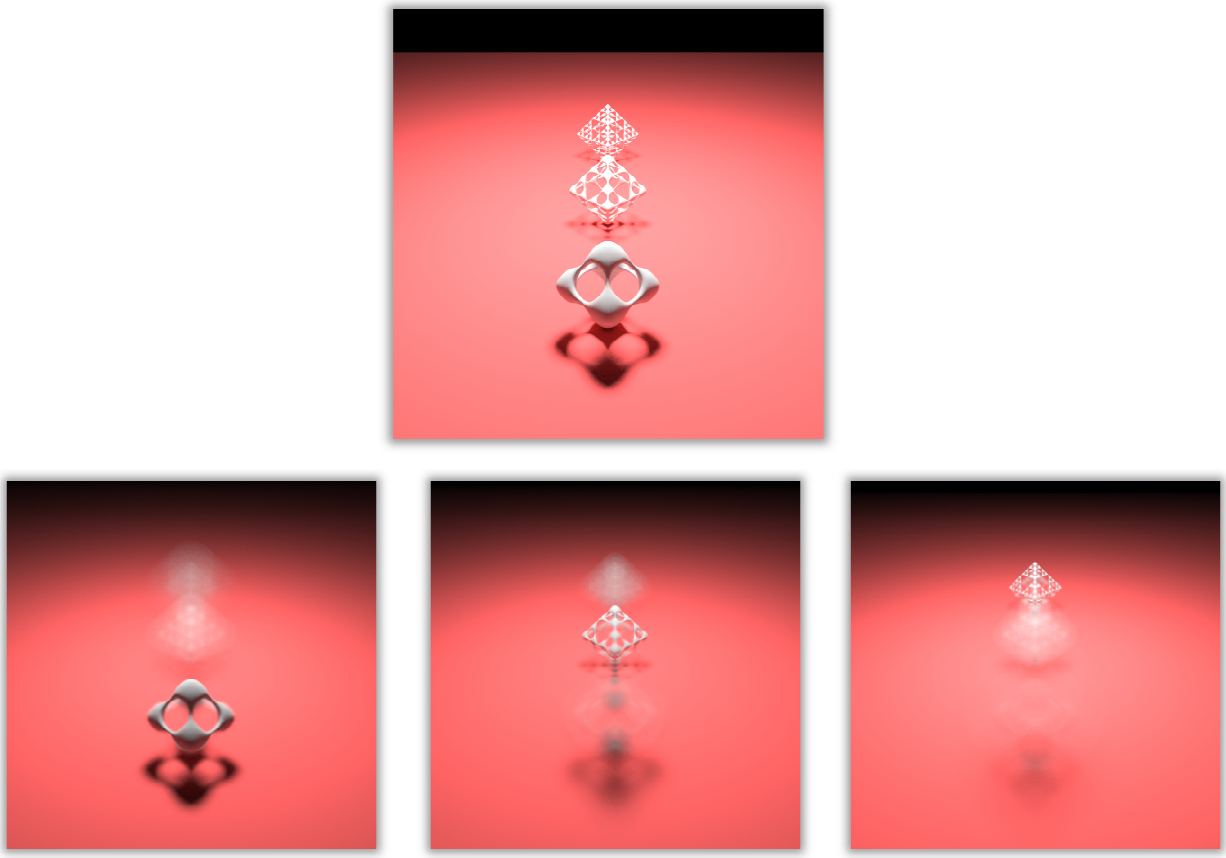

Figure 11: Milton incorporates a thin lens camera model, which supports depth of field via realistic parameters such as aperture and focal distance. The top image shows a simple scene consisting of three fractal meshes, path traced using a pinhole camera (no depth of field). The three images below show the effect of varying the focal distance parameter. The focus in these images was set *automatically* by specifying a focal point in NDC and then calculating the focal distance as the world-space distance between the camera and the first object intersected by a ray shot through the focal point.

Figure 12: Example of dispersion (yellow region), in which the internal index of refraction of an object is wavelength-dependent.

Chapter 5

Conclusion

We have presented Milton as a high quality, open source rendering framework, with a focus on its design and the lessons we have learned along the way. In addition, we have provided an alternative, implementation-focused reference for MLT in an attempt to clarify its subtleties, including an open source implementation available in Milton.

On a broader level, we believe that Milton may be useful as a general educational framework for those wishing to learn the principles of rendering. Milton has already been used successfully as support code for several projects in the graduate-level computer graphics course at Brown University (CS224). Feedback from this trial has been encouraging, given that, we hope Milton to one day fulfill a niche within the educational computer graphics community, of being a free, well-structured open source rendering framework that is both clean, extensible, and capable of advanced rendering techniques such as MLT. In particular, although there are many rendering engines out there, there are none that fulfill all of the goals for Milton. As of writing this document, there are several popular, free, advanced rendering engines available online (e.g., Indigo, PBRT, Luxrender, Yafray, Aqsis, Sunflow, etc.), only some of which are open-source (PBRT, Luxrender, Yafray, and Sunflow), and the only one of which in the author's opinion passes as well-written, well-documented, high quality code, is Sunflow, which is written in Java. In this respect, we believe that although there are many renderers with features similar to Milton available online, there remains an unfilled niche for a clean, well-documented / open-source, global illumination rendering framework written in C++, a niche that we hope Milton may someday grow to fulfill.

Appendix A

Bidirectional Definitions

Let $\bar{y} = y_0 y_1 \ldots y_{k-1}$ be a path composed of k vertices (zero-indexed as in C/C++), where vertex $y_0$is assumed to lie on a light source and vertex y_{k-1} is assumed to lie on the film plane. For reference purposes, we define the following cumulative probability density functions p_s^L and p_t^E, as well as cumulative Monte Carlo estimators of the form f/p, for light subpaths and eye subpaths of a given length, denoted by α_s^L and α_t^E, respectively (as in chapter 10 of Veach's thesis).

$$p_s^L = \begin{cases} 1, & s = 0 \\ p_A(y_0), & s = 1 \\ \overrightarrow{p_{\sigma^\perp}}(y_{s-1}|y_{s-2}) G(y_{s-2} \leftrightarrow y_{s-1}) \cdot p_{s-1}^L, & 2 \le s \le k \end{cases}$$

$$p_t^E = \begin{cases} 1, & t = 0 \\ p_A(y_{n-1}), & t = 1 \\ \overleftarrow{p_{\sigma^\perp}}(y_{k-t}|y_{k-t+1}) G(y_{k-t} \leftrightarrow y_{k-t+1}) \cdot p_{t-1}^E, & 2 \le t \le k \end{cases}$$

$$\alpha_s^L = \begin{cases} 1, & s = 0 \\ \dfrac{L_e^{(0)}(y_0)}{p_A(y_0)}, & s = 1 \\ \dfrac{f_s(y_{s-3} \to y_{s-2} \to y_{s-1})}{\overrightarrow{p_{\sigma^\perp}}(y_{s-1}|y_{s-2})} \cdot \alpha_{s-1}^L, & 2 \le s \le k \end{cases}$$

$$\alpha_t^E = \begin{cases} 1, & t = 0 \\ \dfrac{W_e^{(0)}(y_{k-1})}{p_A(y_{k-1})}, & t = 1 \\ \dfrac{f_s(y_{s-3} \to y_{t-2} \to y_{t-1})}{\overleftarrow{p_{\sigma^\perp}}(y_{k-t}|y_{k-t+1})} \cdot \alpha_{t-1}^E, & 2 \le t \le k \end{cases}$$

In order to gain intuition for what these cumulative functions look like, here are equivalent definitions presented in a more intuitive (though less precise), expanded form:

$$p_s^L = p_A(y_0) \cdot \left(\prod_{i=1}^{s-1} \overrightarrow{p_{\sigma^\perp}}(y_i|y_{i-1}) G(y_{i-1} \leftrightarrow y_i) \right)$$

$$p_t^E = \left(\prod_{i=n-t}^{k-2} \overleftarrow{p_{\sigma^\perp}}(y_i|y_{i+1}) G(y_i \leftrightarrow y_{i+1}) \right) \cdot p_A(y_{k-1})$$

$$\alpha_s^L = \frac{L_e^{(0)}(y_0)}{p_A(y_0)} \cdot \left(\prod_{i=1}^{s-1} \frac{f_s(y_{i-2} \to y_{i-1} \to y_i)}{\overrightarrow{p_{\sigma^\perp}}(y_i | y_{i-1})} \right)$$

$$\alpha_t^E = \left(\prod_{i=n-t}^{k-2} \frac{f_s(y_i \to y_{i+1} \to y_{i+2})}{\overleftarrow{p_{\sigma^\perp}}(y_i | y_{i+1})} \right) \cdot \frac{W_e^{(0)}(y_{k-1})}{p_A(y_{k-1})}$$

The following functions are defined in chapter 10 of Veach's thesis for use in bidirectional path tracing and are provided here both for completeness of this reference and because they will be referred to in *Appendix B*.

$$p_{s,t}(\bar{y}) = p_s^L p_t^E$$

$$c_{s,t} = \begin{cases} L_e(y_0 \to y_1), & s = 0, t > 0 \\ W_e(y_{k-2} \to y_{k-1}), & s > 0, t = 0 \\ f_s(y_{s-2} \to y_{s-1} \to y_{n-t}) G(y_{s-1} \leftrightarrow y_{k-t}) f_s(y_{s-1} \to y_{k-t} \to y_{k-t+1}), & s > 0, t > 0 \end{cases}$$

$$C_{s,t}^* = \alpha_s^L c_{s,t} \alpha_t^E$$

$$w_{s,t} = \frac{p_{s,t}^2}{\sum_{s \geq 0, t \geq 0} p_{s,t}^2} \text{ (power heuristic with } \beta = 2)$$

$$f(\bar{y}) = L_e(y_0 \to y_1) G(y_0 \leftrightarrow y_1) \left(\prod_{i=1}^{k-2} f_s(y_{i-1} \to y_i \to y_{i+1}) G(y_i \leftrightarrow y_{i+1}) \right) W_e(y_{k-2} \to y_{k-1})$$

Appendix B

MLT Acceptance Probability

We present here the explicit acceptance probability used for *bidirectional mutations* in Milton's implementation of MLT, using the same notation as in chapter 11 of Veach's thesis.

Again, let $\bar{y} = y_0 y_1 \dots y_{k-1}$, and let $\bar{x} = x_0 x_1 \dots x_{n-1}$ be paths composed of k and n vertices respectively.

$$a(\bar{y}|\bar{x}) = \min\left\{1, \frac{f_{lum}(\bar{y})T(\bar{x}|\bar{y})}{f_{lum}(\bar{x})T(\bar{y}|\bar{x})}\right\} = \min\left\{1, \frac{Q(\bar{x}|\bar{y})}{Q(\bar{y}|\bar{x})}\right\}$$

$$Q(\bar{y}|\bar{x}) = \frac{T(\bar{y}|\bar{x})}{f_{lum}(\bar{y})}$$

$$= \frac{p_d[l,m]\sum_{i=0}^{k_a} p_a[i, k_a - i]\, p_{l+i,k-(l+i)}(\bar{y})}{f_{lum}(\bar{y})}$$

$$= p_d[l,m]\sum_{i=0}^{k_a} p_a[i, k_a - i]\frac{p_{l+i,k-(l+i)}(\bar{y})}{f_{lum}(\bar{y})}$$

$$= p_d[l,m]\sum_{i=0}^{k_a} \frac{p_a[i, k_a - i]}{C^*_{l+i,k-(l+i)}}$$

We set $p_{l+i,k-(l+i)}(\bar{y})$ to zero if either y_{l+i} or $y_{k-(l+i)}$ is specular.

Bibliography

[DBB03] Dutré, P., P. Bekaert, and K. Bala. *Advanced Global Illumination*. Natick, Massachusetts: A. K. Peters, 2003.

[Cra05] Craiu, R., C. Lemieux. Acceleration of the Multiple-Try Metropolis using Antithetic and Stratified Sampling. 2005.

[Dem02] Demoreuille, P., T. Shaw. A Framework for Monte Carlo Image Synthesis. *Honor's Thesis, Brown University*, 2002.

[Deu03] Dutré, P. Global Illumination Compendium. *www.cs.kuleuven.ac.be/~phil/GI/*, 2003.

[JAF01] Jensen, H. W., J. Arvo, M. Fajardo, P. Hanrahan, D. Mitchell, M. Pharr, and P. Shirley. State of the art in Monte Carlo ray tracing for realistic image synthesis. *SIGGRAPH Course 29*, Los Angeles, 2001.

[Jen01] Jensen, H. W. Realistic Image Synthesis Using Photon Mapping. A K Peters, Natick, Massachusetts, 2001. ISBN 1-56881-147-0.

[Kaj86] Kajiya, J. The Rendering Equation. In *Computer Graphics (SIGGRAPH 86 Proceedings)* (Aug. 1986), vol. 20, pp. 143-150.

[Kel02] Kelemen, C., L. Szirmay-Kalos, G. Antal, F. Csonka. A Simple and Robust Mutation Strategy for the Metropolis Light Transport Algorithm. *Eurographics 2002*.

[KK04] Kollig, T., A. Keller. Illumination in the Presence of Weak Singularities. *Monte Carlo and Quasi-Monte Carlo Methods* (2004), pp. 245-257.

[Laf94] Lafortune, E., Y. Willems. Bi-directional path tracing. In *CompuGraphics Proceedings* (Alvor, Portugal, Dec. 1993), pp. 145-153.

[Liu00] Liu, J., F. Liang, and W. Wong. The Multiple-Try Method for Local Optimization in Metropolis Sampling. In *Journal of the American Statistical Association*; Mar 2000; Vol. 95, No. 449; ABI / INFORM Global p. 121.

[Liu01] Liu, J., F. Liang, and W. Wong. A Theory for Dynamic Weighting in Monte Carlo Computation. *Journal of the American Statistical Association*; June 2001; Vol. 96, No. 454, Theory and Methods.

[Pau00] Pauly, M., T. Kollig, A. Keller. Metropolis Light Transport for Participating Media. In *Rendering Techniques 2000 (Proceedings of the Elevents Eurographics Workshop on Rendering)*, pp. 11-22.

[Pau99] Pauly, M. Robust Monte Carlo Methods for Photorealistic Rendering of Volumetric Effects. *Master's Thesis, Universität Kaiserslautern*.

[Seg07] Segovia, B., J-C., Iehl, B. Péroche. Coherent Metropolis Light Transport with Multiple-Try Mutations. Rapport de recherché RR-LIRIS-2007-015, Soumis à Eurographics Symposium on Rendering 2007.

[Vea94] Veach, E., L. Guibas. Bidirectional Estimators for Light Transport. In *Eurographics Rendering Workshop 1994 Proceedings* (June 1994), pp. 147-162. Also in *Photorealistic Rendering Techniques*, Springer-Verlag, New York, 1995.

[Vea95] Veach, E., L. Guibas. Optimally Combining Sampling Techniques for Monte Carlo Rendering. In *SIGGRAPH 95 Proceedings* (Aug. 1995), Addison-Wesley, pp. 419-428.

[Vea97] Veach, E. Robust Monte Carlo Methods for Light Transport Simulation. *PhD Thesis, Stanford University*, 1997.

[VG97] Veach, E., L. Guibas. Metropolis Light Transport. In *SIGGRAPH 97 Proceedings* (Aug. 1997), Addison-Wesley, pp. 65-76.