Markov Chain Monte Carlo Methods for Global Illumination

Hugh McCabe

October 4, 1999

Image Synthesis Group
Department of Computer Science
Trinity College
Dublin 2, Ireland.
Hugh.McCabe@cs.tcd.ie

Abstract

Monte Carlo methods are techniques for solving difficult numerical problems using random numbers. In recent years they have been applied successfully to various rendering problems in Computer Graphics. In particular they provide an elegant means of solving the Global Illumination problem, that is, correctly calculating the distribution of light within an environment, taking into account all forms of scattering, absorption, and inter-reflection which commonly occur. These solutions have the advantage that they are unbiased and can easily deal with arbitrarily complex geometry. This paper provides an overview of the use of these techniques and in particular concentrates on those which use Markov Chains. A Markov Chain is a sequence of random samples where each sample is probabilistically determined by the previous one in the sequence. This gives rise to a family of rendering algorithms which employ random walks through coordinate space. We describe these algorithms, concentrating on Path Tracing and Bidirectional Path Tracing, detail their implementation, and present results which compare and contrast their effectiveness. We also discuss some recent developments of these ideas which employ adaptive importance sampling to the random walks.

Contents

1 Introduction

A primary goal of Computer Graphics research over the last twenty years has been the automatic generation of images which are as close as possible to photographs of real objects and environments. This is commonly known as *photorealism* or *photorealistic image synthesis*. It has become increasingly apparent that the achievement of this goal is dependent on the ability to simulate the distribution of light within an environment. This involves modeling the emission characteristics of light sources and modeling the complex interaction of light with the atmosphere and the objects in the scene. If we assume the absence of so-called *participating media*, that is media like smoke and fog which significantly affect the transport of light through the atmosphere, then we are left with simulating the interaction of light with surfaces. These interactions take the form of absorption, scattering, and transmission. As well as performing such *local* computations we must also simulate *global* light transport. This means that as well as dealing with the interaction of light which reaches a surface directly from a light source we must also deal with the interaction of light which reaches a surface as a result of scattering or transmission from or through other surfaces.

Early computer graphics imagery tended to ignore global transport issues and compute the local interaction of light with surfaces using simplified reflection models[16][7] .Ray tracing [25] and radiosity [5] added global transport to the simulation. Both techniques did so by making severe assumptions regarding how light can be scattered off surfaces. Subsequent work in rendering was aimed at extending these techniques to handle wider classes of lighting effects, dealing more efficiently with the computational demands, and designing hybrid approaches which aim to combine their advantages.

Monte Carlo methods have been used extensively in recent years, primarily within the context of ray tracing. The primary advantages of these techniques are that they are capable of simulating all types of light transport and that they can deal with arbitrarily complex scene geometry. The main disadvantage is the massive computational demands which ensue. The earliest application was due to Cook[2] who introduced distribution ray tracing in order to solve a variety of previously difficult rendering problems such as motion blur and depth of field. Since then Monte Carlo methods have been applied to a variety of problems with much success. Among the most notable of these have been Shirley and Wang's work on using Monte Carlo Methods to provide physically accurate evaluation of direct lighting[18]; the introduction of path tracing as a means of solving the rendering equation[10]; Dutre's work on light tracing[3]; bidirectional path tracing [14][21]; and Veach's more recent application of Metropolis sampling[24].

In this report we will provide an overview of various Monte Carlo methods for carrying out photorealistic rendering. We will describe these techniques, describe our implementation of these techniques, and present results which clarify the relative advantages and disadvantages of each. The implementation was carried out using the public domain ray tracer RayShade as a core rendering engine. Section 2 describes the image synthesis problem in general, concentrating on an exposition of the rendering equation introduced by Kajiya [10]. We will also discuss how image synthesis techniques like ray tracing and radiosity can be viewed as attempts to solve the rendering equation by making simplifications to it.

Section 3 provides an overview of Monte Carlo methods. We discuss the basic technique of Monte Carlo integration and present some well-known methods of reducing the variance associated with this technique. Section 4 describes how this can then be employed to solve the rendering equation and consequently generate images using *path tracing*. It details the implementation of PT (Path Tracer): the first of three renderers which we present in this report. Section 5 moves on to discuss an important improvement to the basic path tracing algorithm called *next event estimation* which carries out direct sampling of light sources. This section discusses sampling spherical and triangular light sources and presents some sample images produced by our second implementation, PTNEE (Path Tracer with Next Event Estimation), which demonstrate the effectiveness of this technique. Section 6 discusses *bidirectional path tracing* which makes possible the generation of images under difficult lighting conditions which path tracing cannot deal with efficiently. It also describes BPT (Bidirectional Path Tracing); the rendering system which we have implemented based on these ideas. Section 6 discusses Veach's application of Metropolis sampling [24], and outlines future research plans.

2 Global Illumination

Image Synthesis techniques for computer graphics rely on simulating the distribution of light around an environment in order calculate how much light impinges on a particular pixel of the image plane. This distribution of light is governed by the position, strength and nature of the light sources in the scene; by the geometry of the objects in the scene along with their associated properties; and crucially by how the light interacts with the objects. This interaction can take the form of reflection, transmission and absorption, but for most image synthesis applications, reflection is the dominant mode of interaction.

To determine the intensity of light which reaches a particular pixel, we must first determine which point on which object is visible through that pixel (through the application of ray casting or z-buffer for example), and then determine how much light is reflected off this point in the direction of the eye (and hence through the pixel). In order to determine how much light is reflected from the point we must know how much light hits the point and then provide some means of calculating how much of this light is reflected in the appropriate direction.

Traditionally computer graphics applications made two important assumptions. Firstly they assumed that the amount of light which hits the point could be calculated by summing the direct contributions of the light sources and ignoring contributions which may themselves be reflections of light from other surfaces. Secondly they assumed that a simple empirical illumination model such as the Phong model[16] was sufficient to model the reflection characteristics of a surface. The combination of these assumptions led to simple shading models, such as Gouraud shading [6], which were computationally efficient but not very accurate.

More recent treatments at image synthesis eschew simplified illumination models like Phong in favour of bidirectional reflectance distribution functions (BRDF's), and attempt to model all inter-reflections of light within an environment. The combination of these concepts is known as global illumination and can be succinctly expressed in mathematical form by the rendering equation [10].

2.1 Bidirectional Reflectance Distribution Functions

The fundamental quantity of image synthesis is radiance which can be defined as the power arriving at or leaving from a surface dA in a certain direction, per unit solid angle $d\omega$ around that direction, per unit projected area perpendicular to the direction of travel. We denote this by L and it can be calculated as:

$$L = \frac{d^2\Phi}{d\omega dA \cos\theta} \tag{1}$$

where Φ is the radiant flux, ω the solid angle, and θ is the angle between the surface normal and the direction of travel under consideration. Intuitively we can think of radiance as perceived intensity. A bidirectional reflectance distribution function (BRDF), f_r, is a function that is associated with a surface, and when given an incoming radiance returns an outgoing radiance. Intuitively this means that given a surface and given a light source, we can use the BRDF of the surface to calculate the amount of light reflected from the surface in any direction as a result of incident light from that light source. We denote this formally as:

$$f_r(x, \Theta_i, \Theta_o) \tag{2}$$

where x is a point on the surface, Θ_i is the incoming direction, and Θ_o is the outgoing direction. The Phong local illumination model can be seen as a simplified version of a BRDF. The Phong model tells us that for all outgoing directions except those close to the specular reflection direction the reflected radiance is constant (diffuse reflection). For an outgoing direction close to the specular reflection direction the outgoing radiance is a function of the angle between this direction and the specular reflection direction. In reality the behaviour of light as it reflects off a surface is far more sophisticated than this and so the added expressive power of a BRDF is necessary.

2.2 The Rendering Equation

The above discussion gives us no insight into how the incoming radiance is actually calculated and this problem forms the real basis of most image synthesis research. As explained previously traditional computer graphics techniques calculated incoming radiance by measuring the direct contributions from light sources and leaving it at that. Let us refer to these light sources as primary emitters. This is not sufficient as these contributions form only a part, albeit a particularly significant part, of the incoming radiance. We must also include incoming radiance that results from reflections off other surfaces. We refer to these other surfaces as secondary emitters. We can theoretically do this by using the BRDF's of the secondary emitters to sum all these contributions. This gives us an integral equation to evaluate. However the input into the BRDF for any point on a secondary emitter is the incoming radiance over the hemisphere of all incoming directions and hence is another integral equation which itself is potentially a combination of primary and secondary emitters. We therefore have a potentially infinitely recursive integral equation to evaluate. This is known as the rendering equation and can be formulated as[1]:

$$L(x, \Theta_x) = L_e(x, \Theta_x) + \int_{\Omega_x} L(y, \Theta_y) f_r(x, \Theta_y, \Theta_x) \, |\Theta_y . N_x| \, d\omega_y \tag{3}$$

where $L(x, \Theta_x)$ is the radiance from point x in some direction Θ_x, in other words, if x is some point in space that we are looking at, then this expresses the radiance coming from this point in the direction of the viewpoint and therefore is the radiance we are interested in calculating. $L_e(x, \Theta_x)$ is the emitted surface radiance in direction Θ_x at point x. This will be zero if the surface in question is not a primary emitter. The integral part of the equation expresses the radiance contributions of the secondary emitters. It is an integral over all the hemisphere of directions above the surface at point x, denoted by Ω_x. This represents all the possible incoming directions and for each one we need to calculate what the incoming radiance is, sum them together, and use the BRDF of the surface to calculate the outgoing radiance in the direction Θ_x. In order to calculate the incoming radiance from point y along direction Θ_y we need to solve an integral equation of the same form as the above and hence the whole process is recursive.

Traditional image synthesis techniques simplify the equation above in order to make it com-

[1] There are many different mathematical formulations of this material. In this report we choose to follow most of the conventions used by Lafortune [14]

putationally tractable. In classical ray tracing [25] the BRDF is replaced with the simple Phong model and the hemisphere of directions along which incoming radiance is calculated is replaced by a finite set of directions. This finite set is comprised of the set of directions towards visible light sources, each of which is handled as a direct lighting term, and the specular reflection direction, which is handled recursively. Infinite recursion is avoided by placing a limit on how far the process can recurse.

Radiosity methods typically avoid the integral by treating the environment as a discrete set of patches rather than a continuous set of points. This means that the integral is replaced by a summation operator and we are left with a matrix of simultaneous equations, which can be solved using a method like Gauss-Seidel [5]. Once more the BRDF is discarded and radiosity operates under the assumption that all surfaces are perfectly diffuse Lambertian emitters and reflectors.

Both of these techniques suffer serious limitations regarding the environments that they are able to handle successfully. Radiosity effectively ignores specular reflection completely and concentrates on modeling diffuse inter-reflections, while ray tracing models specular reflection and inter-reflection successfully but is limited to only direct lighting for the diffuse component of a surface. In their basic form, both methods fail to deal with the spectrum of reflection phenomena which lie between the specular and diffuse poles since they use simplified BRDF's. However, many extensions and hybrid techniques have since been developed in order to extend the range of situations that can be handled. In this report we concentrate on Monte Carlo methods and show how they potentially provide a means of solving the rendering equation completely and hence provide a basis for implementing physically accurate lighting simulation and rendering systems.

2.3 RayShade

RayShade is a public domain ray tracer that was written by Craig Kolb. It uses a simple scene language that includes support for a variety of primitive objects (boxes, planes, spheres, cones, polygons etc.) and allows the user to combine these together using CSG operations. It provides a variety of different light sources (point, area, extended) but, for reasons we explain in section 4, deals with non-point light sources by treating them as collections of point light sources. It includes support for texturing. RayShade was chosen as a base system to build on for the purposes of this research for the following reasons

- The source code is freely available and researchers are invited to use it.
- The code is structured in such a way that extending it is possible.
- It includes many optimisations, such as automatic calculation of bounding boxes, which make it a fast and efficient system.

The pseudo-code below illustrates the central driving functions of RayShade and also serves to explain the basic ray tracing mechanism. The central function is called ShadeRay and this is initially given a ray which represents an eye-ray through the pixel and information about the intersection of this ray with the environment. If this intersection is valid it computes surface properties and normals at the intersection point and calls a Shade function to compute the colour at that surface. Shade then calls functions to carry out direct lighting calculations, determines whether we have reached the maximum depth to which we are willing to trace and if not, computes the reflection angle and calls ShadeRay recursively to trace further rays in those directions. The important thing to note is that it follows rays backwards according to the specular reflection angle. RayShade also deals with transparent objects by following transmitted rays backwards and computing the light transport along these paths also. For the moment we are going to ignore the issues involved in transmission in order to simplify the presentation of the material. Further information about the standard ray tracing algorithm can be found in [25]

```
ShadeRay(Ray,Intersection)
    If(Intersection not valid)
        return black
    Compute Surface Normal and Surface Properties
    Colour=Shade(Intersection)
    return(Colour)
End
Shade(Intersection)
    DirectColour=LightRay(Intersection)   //does direct lighting computation
    If(reached maxdepth)
        return DirectColour
    If(reflective)
        RefColour=ReflectRay(Intersection)
    return(DirectColour+RefColour)
End
LightRay(Intersection)
    IncomingLight=Direct Light from Light Sources
    Colour=IncomingLight * Phong Model
    return Colour
End
ReflectRay(Intersection)
    Choose new Ray              // using the specular reflection angle
    Intersection=TraceRay(Ray)    // Find intersection point
    Colour=ShadeRay(Ray,Intersection)
    return(Colour)
End
```

3 Monte Carlo Methods

Monte Carlo methods are techniques for computing answers to problems using the properties of random numbers. The first known application of Monte Carlo methods was an experiment carried out by Hall in 1873 to compute the value of π by repeatedly throwing a needle on a piece of paper with parallel lines drawn on it. They were extensively used during World War II by von Neumann to carry out simulations related to atomic research and subsequently, during the 50's and 60's, were applied to a wide variety of physical and mathematical problems. They can be used to carry out simulations of problems which themselves have an underlying probabilistic nature or they can be applied to deterministic problems for which solutions are impossible or difficult to obtain using analytical and numerical techniques.

3.1 Monte Carlo Integration

The most common application of Monte Carlo methods is as a numerical means of evaluating integrals and this application has the closest relevance to the problem at hand. Monte Carlo integration proceeds as follows. Suppose we have some function that we wish to integrate over the range [0,1]. In other words we want to evaluate

$$I = \int_0^1 f(x)dx \tag{4}$$

We can estimate the value of this integral by taking a uniform random number r over $[0, 1]$ and evaluating $f(r)$.

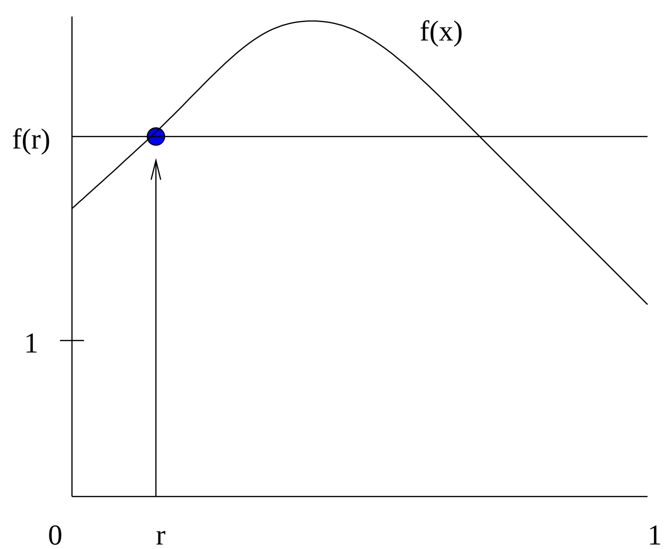


Figure 1: A primary estimator for the integral.

This is called a primary estimator for the integral. When evaluated for a specific value of r it forms an estimate for the value of the integral and is calculated as the area of the rectangle under the evaluated point (see Figure ??). The basis of Monte Carlo Methods is to make this estimate more accurate by taking numerous such primary estimators and combining them into a secondary estimator by averaging them. So the secondary estimator is the average of N primary estimator (Figure 2)

$$I_{sec} = \frac{1}{N} \sum_{i=1}^{N} f(r_i) \tag{5}$$

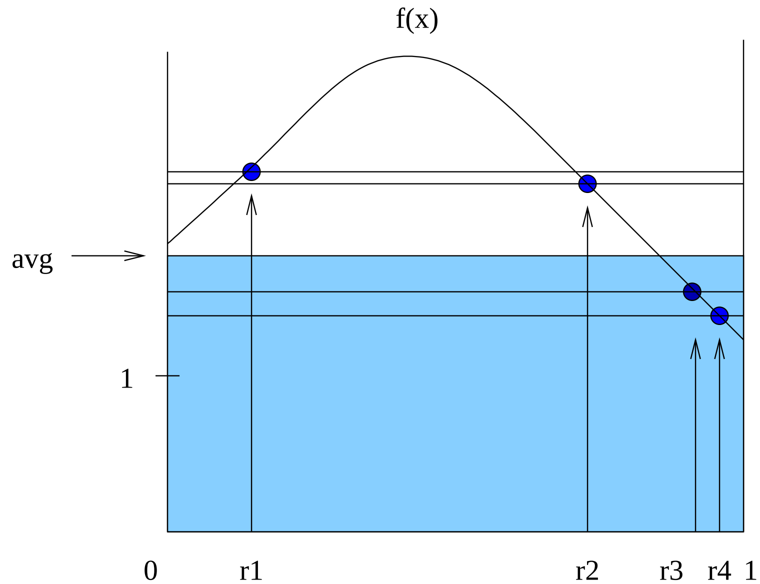


Figure 2: Secondary estimator calculated by averaging four primary estimators. As we can see we don't get a very good estimate because we have failed to sample important regions of the integral.

The accuracy of the result depends on how many samples are taken. If we use uniform random samples then we get an unbiased result. This means that as the number of samples approaches infinity the answer converges to the correct result. It can be shown that, if N is the number of primary estimators, then the variance of the secondary estimator is reduced by a factor of $\frac{1}{N}$. In order to significantly reduce the variance of the secondary estimator, it is more productive to try to reduce the variance of the primary estimators. This is largely dependent on the distribution of samples. For example if all values of r which are chosen are clustered at one end of the domain and that end of the domain is not that significant then we won't get a good result. There are various ways of countering this and hence reducing the variance. Stratified sampling splits the domain into sub-domains and chooses one sample at random within each sub-domain so that the samples are spread more evenly over the domain as a whole. Importance sampling distributes samples around regions of the domain that are significant.

3.2 Improving the Estimate

The simplest form of stratified sampling divides the domain into a number of sub-domains and estimates the partial integrals associated with each of these sub-domains with standard Monte Carlo integration. Figure 3 shows how this works.

The variance of stratified sampling is always less than or equal to that of the ordinary secondary estimator with the same number of samples. Stratified sampling is also unbiased. The cost

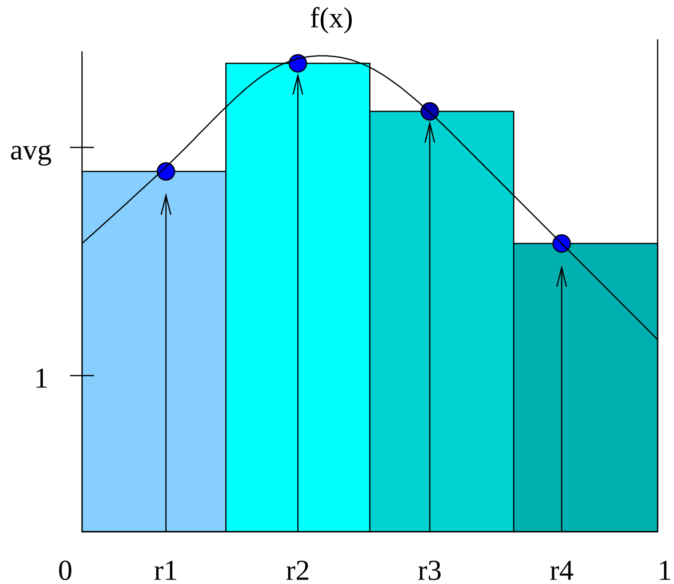


Figure 3: Stratified sampling: Note that we obtain a better result with the stratified samples than with uniform ones in this example.

of distributing the samples in this manner is relatively small so stratified sampling can often be successfully applied. Distributed ray tracing employs a form of stratified sampling in two dimensions when sending pixel rays into a scene. It divides the pixel up into a grid and sends one ray, randomly placed, through each grid element. Stratified sampling can be extended to two or more dimensions easily by using techniques such as N-rooks sampling.

Importance sampling attempts to reduce variance by taking more samples from regions of the function which have higher values and hence affect the value of the integrand more than those with low values. In order to explain how this is done we quickly review some concepts from probability theory.

Associated with a random variable y is a probability distribution function $P(y)$. This gives the probability that the value of the random variable is less than or equal to y. For example, if y is a random variable, which describes the outcome of rolling a die then, if y is 6, $P(y)$ is 1. If y is 1 then $P(y)$ is $1/6$ and so on. Associated with the probability distribution function is the probability density function (PDF) $p(y)$. If $P(y)$ is continuous over the domain of the random variable then the PDF is the derivative of the distribution. The PDF allows us to calculate the following. Given two values a and b, where $a <= b$ and both lie within the domain of the random variable, then integrating the PDF between this range gives us the probability that the random variable will fall within it. More formally:

$$P[a < y \leq b] = \int_a^b p(x)dx \tag{6}$$

We carry out importance sampling by distributing samples according to a PDF that has a roughly similar shape to the integrand. So for example if we look at the function in Figure 4 we see that distributing samples according to the PDF will give us more samples in important regions of the domain.

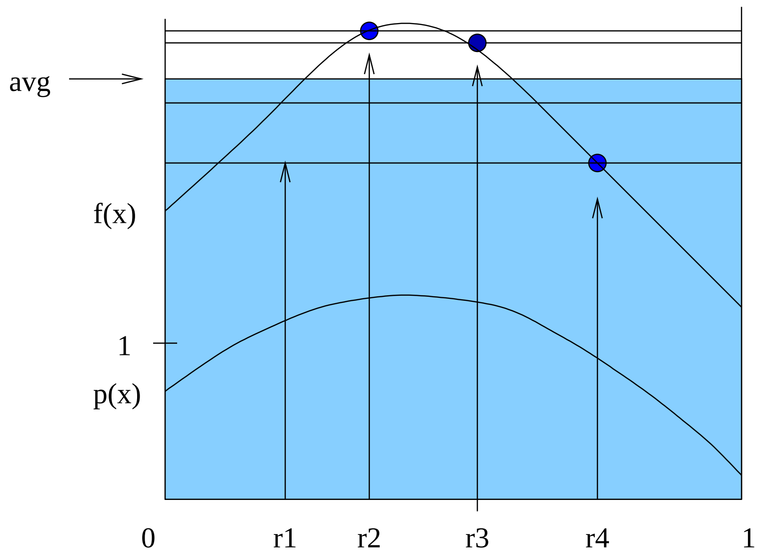


Figure 4: Importance sampling: samples are distributed to important regions of the integral and hence we get a better estimate.

This implies that we have some method of distributing the samples according to the relevant PDF. This can be done by using the inverse of the probability distribution function, $P(x)$, associated with the PDF, $p(x)$. So we choose a random sample r, then we distribute it according to the PDF by calculating $x = P^{-1}(r)$. This is represented in Figure 5.

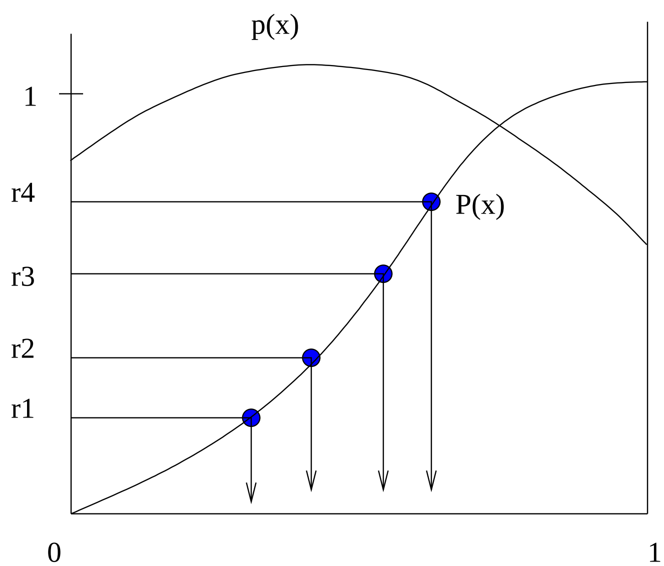


Figure 5: Inverting p(x)

This however introduces bias. This means that the answer converges, but not necessarily to the correct result. We have to adapt the estimator so that this is countered correctly. The original integral is re-written as:

$$I = \int_0^1 \frac{f(x)}{p(x)} p(x) dx \tag{7}$$

If a random variable r is sampled according to the PDF $p(x)$ over $[0, 1]$ the primary estimator is:

$$I_{imp} = \frac{f(r)}{p(r)} \tag{8}$$

The secondary estimator then becomes:

$$I_{impsec} = \frac{1}{N} \sum_{i=1}^{N} \frac{f(r)}{p(r)} \tag{9}$$

The success if this approach is largely dependent on being able to find a PDF which is as close as possible in shape to the function being integrated and which meets the following requirements.

- $p(x) > 0$ for each $x \in [0, 1]$ for which $f(x) \neq 0$,
- $\int_0^1 p(x)dx = 1$,
- It is possible to invert the cumulative probability distribution function in order to distribute samples according to $p(x)$.

To see an example of this working in practice suppose we wish to integrate the function $f(x) = 2x^3$. We can carry out this operation by using conventional analytical techniques and achieve the following result.

$$\int_0^1 2x^3 = \frac{1}{2}(1)^4 - \frac{1}{2}(0)^4 = \frac{1}{2} \tag{10}$$

Suppose we choose to try and solve this using Monte Carlo also. We start by using straight Monte Carlo with no variance reduction techniques. We select four random points on the interval $[0, 1]$.

$$r_i = [0.2, 0.3, 0.5, 0.8] \tag{11}$$

This will give us:

$$I = \int_0^1 f(x)dx = \frac{1}{4} \sum_{i=1}^{4} f(r_i) = \frac{1}{4}(0.016 + 0.054 + 0.25 + 1.024) = 0.336 \tag{12}$$

This is not a particularly good estimate but the unrealistically low number of samples accounts for this.

We now try importance sampling using the same four sample points. We choose $p(x) = 3x^2$ as our PDF. It is straightforward to verify that it meets the requirements for valid PDF's as outlined above. The cumulative probability distribution function $P(x)$ is calculated by integrating the PDF.

$$P(x) = \int_0^x p(t)dt = x^3 \tag{13}$$

This function is easily inverted allowing us to generate new sample points as follows:

$$x_i = P^{-1}(r_i) = (r_i)^{\frac{1}{3}} \tag{14}$$

So, using the same initial random points as before, we generate the set:

$$x_i = [0.584, 0.669, 0.793, 0.928] \tag{15}$$

Applying Monte Carlo integration using these sample points gives us:

$$I = \int_0^1 f(x)dx = \frac{1}{4}\sum_{i=1}^{4}\frac{f(x_i)}{p(x_i)} = \frac{1}{4}(\frac{0.398}{1.023} + \frac{0.598}{1.342} + \frac{0.997}{1.886} + \frac{1.598}{2.583}) = 0.4954 \tag{16}$$

This is much closer to the actual answer as calculated analytically by Equation 10 than the result obtained by "blind" Monte Carlo in Equation 12. Of course this accuracy is purely a consequence of choosing a PDF which was close to the function we are trying to integrate. Successful application of importance sampling is dependent on choosing a useful PDF.

So we have seen that Monte Carlo methods provide a feasible means of solving integral equations. They are by no means the only way of doing so however and before finishing we should discuss why we favour these methods over deterministic numerical techniques. The most important reason is that the convergence rate of Monte Carlo methods is not affected by the dimensionality of the integral. The integrals we will be dealing with exhibit high dimensionality. The fundamental quantity of radiance itself is a five-dimensional function since it is defined at a point in three-dimensional space and for a particular direction. We will also be sampling over pixels which add a further two dimensions and if we include effects such as motion blur and depth of field then the dimensionality of the problem increases further. Deterministic techniques do not, in general, perform well when the dimensionality is increased. Also, Monte Carlo methods provide the means to compute unbiased solutions, and the means to quantity the level of error in the image. The interested reader should consult [11] for more information about Monte Carlo methods.

4 Pure Path Tracing

Ray tracing was the first significant attempt to model light transport effectively [25]. In this technique rays are traced backwards through the scene by following the specular reflection angle. This means that it only computes specular reflection globally. Local diffuse calculations are done by directly sampling the light source at each point on the path. Consequently a wide variety of light transport situations are not handled. A significant improvement on this situation resulted from the introduction of *distribution ray tracing* [2]. This used Monte Carlo methods to sample pixels, light sources and ray directions resulting in the addition of important effects such as *depth of field* and *soft penumbrae.* However the lack of a complete solution to the rendering equation meant that some significant lighting effects were still not possible. For example *colour bleeding,* which is the result of light being diffusely reflected from one surface to another, and *caustics,* which are the results of light being specularly reflected from or transmitted through, one surface, and then diffusely reflected towards the eye by another surface.

Path tracing [10] provides a means of solving the rendering equation completely. It can be easily understood as a generalisation of the classic Whitted-style ray tracing algorithm. In Whitted ray tracing we follow rays backward according to the specular reflection angle. This means that we are essentially following specular-specular light transport chains. In path tracing, we follow the same general algorithm, but we choose random directions to follow at each intersection point. By doing so we are constructing so-called *random walks* through the scene and computing the light transport along these walks. In the following sections we give mathematical details as to why this constitutes a correct solution of the rendering equation, describe PT, our implementation of these ideas, and present appropriate results.

4.1 Monte Carlo Integration of the Rendering Equation

The Rendering Equation is a *Fredholm Integral of the Second Kind.* Such equations take the general form shown below:

$$f(x) = g(x) + \int_0^1 K(x,y)f(y)dy \tag{17}$$

Here $g(x)$ is a known function, $f(x)$ is the unknown function, and $K(x,y)$ is known as the Kernel operator. As we saw in the previous section we can estimate the values of integral equations using Monte Carlo methods. In the above case we can compute $f(x)$ for some given x but this requires us to compute $f(y)$, which is an equation of the same form, i.e.

$$f(y) = g(y) + \int_0^1 K(y,y')f(y') \tag{18}$$

This in turn requires us to compute $f(y')$ in the same manner. So, the process is recursive. At each step we take a sample ε_i over the interval $[0,1]$ according to some PDF $p_i(x)$. So we get a primary estimator as follows:

$$I = \sum_{i=0}^{\infty} \left[\prod_{j=1}^{i} \frac{K(\varepsilon_{j-1}, \varepsilon_j)}{p_j(\varepsilon_j)} \right] g(\varepsilon_i) \tag{19}$$

where $\varepsilon_0 = x$. Our series of samples, $\varepsilon_1, \varepsilon_2, ...$ is called a *Markov Chain* because the PDF for each point depends only on the previous point. This is also known as a *Random Walk*. The average of many such random walks forms a secondary estimator for $f(x)$. In practice we cannot extend a random walk to infinity so we need some method of terminating the walk. This cannot be done by simply terminating at some fixed number of recursions because this introduces bias. The usual way to deal with this situation is to employ some form of *Russian Roulette*. This means that the paths are terminated probabilistically and results in an unbiased estimator.

So how does this apply to the Rendering Equation? Recall the equation from section 2.2:

$$L(x, \Theta_x) = L_e(x, \Theta_x) + \int_{\Omega_x} L(y, \Theta_y) f_r(x, \Theta_y, \Theta_x) \left|\Theta_y . N_x\right| d\omega_y \tag{20}$$

Ω_x represents the hemisphere of incoming directions around x. The Θ terms all represent directions and the f_r function is the BRDF. As we can see this equation has the same form as 17 and therefore we can solve it in the same manner. Before doing so we need an expression which characterises the radiant flux seen through a pixel from a particular point (usually the viewpoint). This can be written as:

$$\Phi = \int_A \int_{\Omega_x} L(x, \Omega_x) W_e(x, \Omega_x) \left|\Omega_x . N_x\right| d\omega_x d\mu_x \tag{21}$$

In this expression A is the set of surfaces in the scene, Ω_x is the hemisphere of directions above the point x, and $W_e(x, \Omega_x)$ is a function which returns 1 for pairs of points and directions that belong to the set of interest and 0 otherwise. In our case the set of interest are all points x on the pixel in question and directions which pass through those points x, reach the viewpoint, and hence contribute to the image. This set is shown in diagram **??**. Evaluating this integral can be

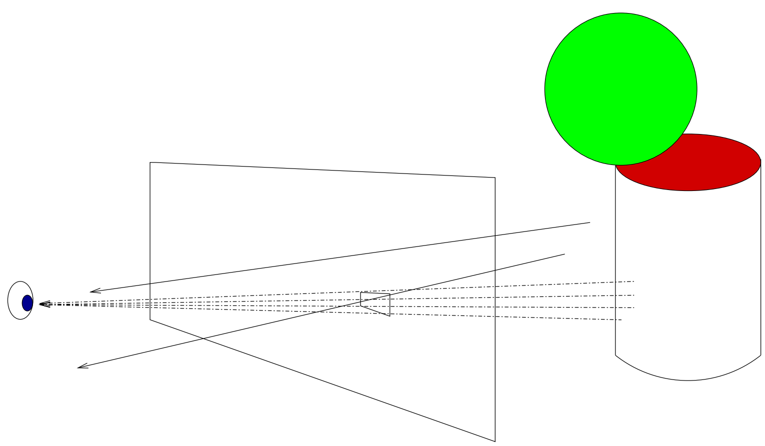


Figure 6: The dotted lines represent points (on the pixel) and directions for which the W function would return 1. The solid lines represent points and directions for which it would return 0. In one case this is because the point is not within the pixel. In the other case it is because the direction does not reach the viewing point and hence will not contribute to the image.

done using the following primary estimator:

$$\langle\Phi\rangle_{path} = \frac{W_e(x_0, \Theta_{x_0}) \left|\Theta_{x_0} . N_{x_0}\right|}{p_0(x_0, \Theta_{x_0})} \langle L(x_0, \Theta_{x_0})\rangle_{path} \tag{22}$$

The method by which this integral is sampled largely depends on the camera model being used in the renderer. Typically it is done by choosing some random point x_0 on the pixel according to the PDF $p_0(x_0, \Theta_0)$ and extending a ray through this point. The PDF can take any form and can be used to implement anything from uniform sampling on the pixel to stratified sampling to Poisson disk style sampling and so on. See Shirley's Ph.D. thesis for a comprehensive discussion of these matters [17].

We can then estimate $L(x_0, \Theta_{x_0})$ by treating it as a recursive integral as explained previously. In practice this amounts to extending a ray at each intersection point and calculating its next intersection point with the scene. By doing this we build up a random walk through the scene. At each intersection point we sample a new direction according to some PDF, $p_i(x_i, \Theta_i)$, over the hemisphere. These walks will contribute to the pixel in question if they eventually reach a light source. We generally terminate the walks by using Russian Roulette and if the walks don't hit light sources then their contribution will be zero. We form a secondary estimator for the pixel by using the average of many such walks.

4.2 Implementation

Path Tracing is relatively straightforward to implement as an extension to a ray tracer like RayShade however there are a number of issues which have to be addressed. Firstly, as mentioned before, the crucial difference between path tracing and ray tracing is that random directions are chosen when recursing backwards rather than a fixed direction (the specular reflection angle). By choosing random directions we are sampling the hemisphere of directions above the point in question. We can simply sample directions uniformly about the hemisphere however we will obtain better results if we employ some form of *importance sampling* whereby we choose directions which are likely to contribute significantly to the result. Secondly, we need to decide on appropriate BRDF's for the surfaces. One option is to simply use the Phong model however Eric Lafortune introduced a modified Phong Model which is more appropriate to physically based rendering [13]. We will leave both these issues aside for the moment and discuss how light transport is computed along a path. Recall the rendering equation as expressed in 20:

$$L(x, \Theta_x) = L_e(x, \Theta_x) + \int_{\Omega_x} L(y, \Theta_y) f_r(x, \Theta_y, \Theta_x) \, |\Theta_y.N_x| \, d\omega_y \tag{23}$$

From now on we will adopt the convention that x_0 denotes the first point on the path (i.e. the point on the pixel) and Θ_{x0} denotes the direction towards this point to the the viewpoint (i.e. the initial ray direction in reverse). Let us also denote $|\Theta_y.N_x|$as $cos\Theta_x$. We can now rewrite the rendering equation as:

$$L(x_0, \Theta_{x0}) = L_e(x_0, \Theta_{x0}) + \int_{\Omega_{x0}} L(y, \Theta_y) f_r(x_0, \Theta_y, \Theta_{x0}) \, |\Theta_y.N_{x0}| \, d\omega_y$$

If we trace a path backwards and hit a light source after n bounces (in PT, and the subsequent renderers, we assume that surfaces are either light sources or light reflectors but not both and hence that we terminate paths when we hit light sources) then we can express the light contribution as:

$$L_{PT}(x_0, \Theta_{x0}) = \left(\prod_{i=1}^{n} \frac{f_r(x_i, \Theta_{xi}, \Theta_{xi-1}) cos\Theta_{xi}}{pdf(\Theta_{x_i})} \right) L_s \tag{24}$$

The L_s term is the radiance of the light source[2] which was struck after n bounces and the $pdf(\Theta_{x_i})$ term is the PDF which was used to sample the hemisphere above the point x_i. An example of this is shown in figure 7. If we expand the formula above for two bounces we get the following:

$$L_{PT}(x_0, \Theta_{x0}) = \frac{f_r(x_1, \Theta_{x1}, \Theta_{x0}) cos\Theta_{x1}}{pdf(\Theta_{x_1})} . \frac{f_r(x_2, \Theta_{x2}, \Theta_{x1}) cos\Theta_{x2}}{pdf(\Theta_{x2})} . L_s \tag{25}$$

It remains to describe how the $BRDF$ and PDF are determined.

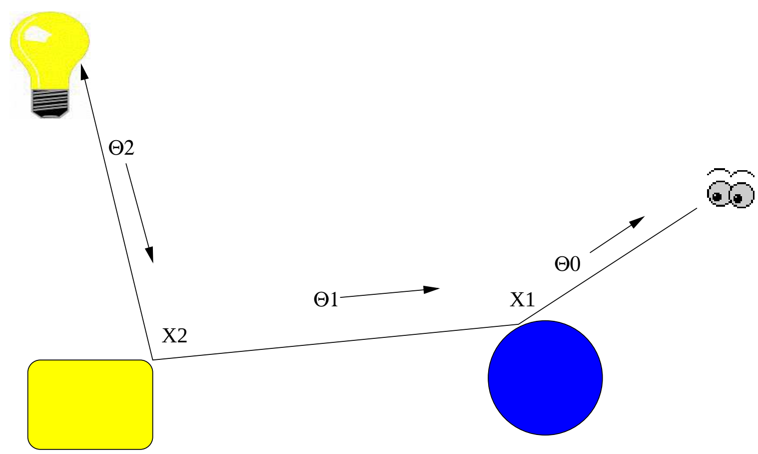


Figure 7: A path with n=2 (i.e. two bounces)

For PT we have used a $BRDF$ which simulates pure Lambertian diffuse reflection only. Such a surface reflects light equally in all directions. Surfaces are assigned a *diffuse reflection coefficient*, k_d[3], and the $BRDF$ becomes:

$$f_r(x, \Theta_i, \Theta_o) = \frac{k_d}{\pi} \tag{26}$$

This is different to the usual Phong style diffuse reflection formula ($k_d cos\theta$) in two ways. Firstly there is no cosine term. This is because the cosine term in the Phong style formula is intended to model how much light reaches the point in question and hence is related to the angle θ between the light source and the point. We don't need this because a $BRDF$ is used in a context whereby we already know what the incoming radiance along Θ_i is. Secondly the k_d term is divided by π. This is because k_d represents the total hemispherical reflectance, i.e. the proportion of the incoming light over the entire hemisphere which is reflected out over the entire hemisphere. So

[2]The L_s term represents the radiance of the light source as seen along the direction in question. . If we denote the total radiance of the light source by L_t then for spherical light sources $L_s = L_t$. However for triangular or area light sources we need to include a cosine term which represents the angle α between the direction from the light source and the surface normal. So in this case $L_s = L_t \cos \alpha$.

[3]This discussion assumes that we are computing monochromatic images. In practice we will want to compute colour images so we use the RGB colour model. So we will have three diffuse reflection coefficients, k_{dr}, k_{dg}, and k_{db}, and carry out three seperate independent calculations at each point. It may also be the case that we are doing full spectral rendering in which case we will effectively have co-efficients for a whole range of wavelengths.

when we integrate the $BRDF$ over the hemisphere we should get k_d. The division by π ensures that we do.

The PDF which we use to sample the hemisphere is a simple cosine distribution. This is intended to ensure that directions which are perpendicular to the surface are favoured over those at oblique angles to the surface:

$$pdf(\Theta) = \frac{\cos\theta}{\pi} \tag{27}$$

Θ is the angle between the direction Θ and the surface normal. If we can sample according to this PDF then the probability of sampling a direction where θ is near 90^o is very small. Lafortune [13] explains how this is done using the following formula. θ is the polar angle with the surface normal and ϕ is the azimuthal angle:

$$(\theta, \phi) = (\arccos\sqrt{\varepsilon_1}, 2\pi\varepsilon_2) \tag{28}$$

We can express this in terms of vector coordinates as:

$$(x, y, z) = (\sqrt{1-\varepsilon_1}\cos(2\pi\varepsilon_2), \sqrt{1-\varepsilon_1}\sin(2\pi\varepsilon_2), \sqrt{\varepsilon_1}) \tag{29}$$

Now if we replace the appropriate terms from equation 26 and equation 27 into equation 24 we get:

$$L_{PT}(x, \Theta_x) = \left(\prod_{i=1}^{n} kd_i\right) L_s \tag{30}$$

Equation 30 is then the basic calculation needed to do pure path tracing with diffuse surfaces.

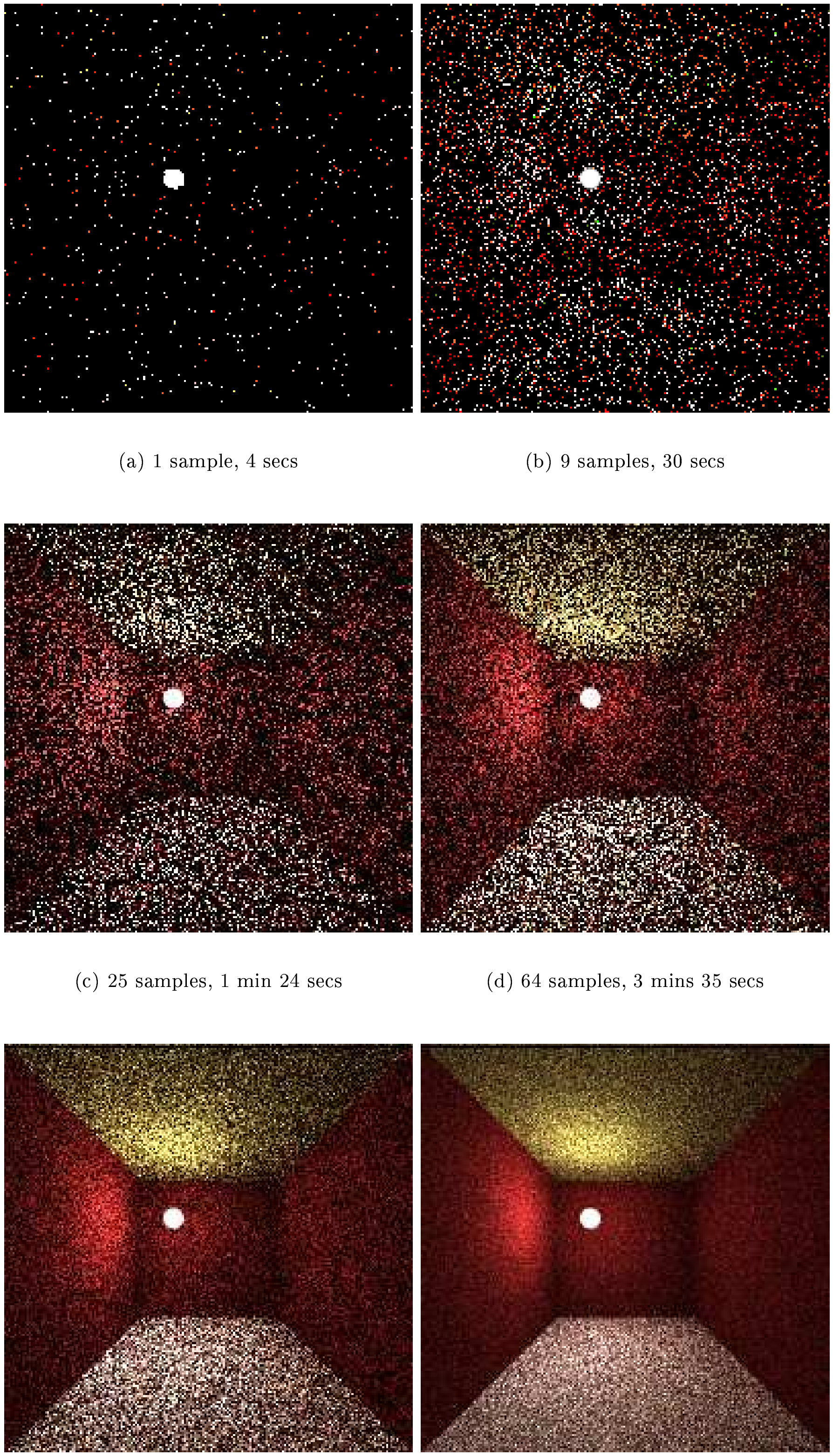
It is instructive to now look at the pseudo-code for PT in order to see how the basic rendering loop differs from that of the ray tracer.

```
ShadeRay(Ray,Intersection)
    If(Intersection not valid)
        return black
    Compute Surface Normal and Surface Properties
    Colour=Shade(Intersection)
    return(Colour)
End
Shade(Intersection)
    If(reached maxdepth)
        return black
    NewDir = GetDiffuseDirection()    // using equation29
    RefColour=ReflectRay(NewDir)
    return(RefColour * (Modified Phong/PDF))  // this is k_d (see equation 30)
End
ReflectRay(Dir)
    Intersection=TraceRay(Dir)    // Find intersection point
    If(Intersection is a Light Source)
        Colour=Direct Light from Light Source
        return(Colour)
```

```
    Else
        Colour=ShadeRay(Ray,Intersection)
        return(Colour)
End
```

4.3 Results

The images in figure 8 were computed using PT. All of the surfaces are diffuse and there is one spherical light source present. Each image is labelled with the number of samples per pixel which were used to create it and the computation time involved. All images were computed with a maximum path depth of 5. We should first note that the image computed with only one sample per pixel is mostly black but with some pixels very bright. This is because the probability of hitting a light source with a path is very low. Most paths which are extended through the pixel don't return anything and hence they are black. However, consider a pixel whose path does hit a light source. This returns a very large radiance value and since this constitutes the sole sample for that pixel it appears extremely bright in the image. As we increase the number of samples per pixel more pixels have paths which hit light sources , so there won't be as many totally black ones, and also the results returned by the pixels will be averaged, the brightness is scaled down, and we begin to discern colours. The number of samples we have to take in order to produce a reasonable image is inordinately large as figure 8 makes clear. The situation improves somewhat when the size of the light source(s) is increased since this means that more paths will hit light sources however pure path tracing, as implemented in PT, does not really constitute a feasible rendering technique.



(a) 1 sample, 4 secs

(b) 9 samples, 30 secs

(c) 25 samples, 1 min 24 secs

(d) 64 samples, 3 mins 35 secs

(e) 256 samples, 14 mins 23 secs

(f) 1024 samples, 57 mins 27 secs

Figure 8: Pure path tracing images

5 Path Tracing With Next Event Estimation and Importance Sampling

The obvious way in which PT can be improved is to incorporate some form of direct sampling of light sources at all points along the path. This section firstly describes a mathematical formulation of this idea within the context of Monte Carlo integration and then details how direct lighting can be carried out in a physically correct manner with a number of different types of light sources. We then move on to discuss importance sampling within the context of path tracing which allows us to probabilistically sample directions along which significant light transport is likely to occur. This allows us to incorporate a full specular/diffuse BRDF. We finish by describing our implementation of these techniques and presenting our results.

5.1 Next Event Estimation

Estimating an integral of the form

$$f(x) = g(x) + \int_0^1 K(x,y)f(y)dx \tag{31}$$

can result in high variance if the function $g(x)$ is mostly 0 over the domain. This is exactly what happens when we evaluate the rendering equation using Monte Carlo integration since the function $g(x)$ represents the emitted radiance at point x . This will only be non-zero if x is on a light source. We can improve this situation by splitting the integral and evaluating each part of it differently.

$$\begin{aligned}
f(x) &= g(x) + \int_0^1 K(x,y)f(y)dy & (32)\\
&= g(x) + h(x) & (33)\\
h(x) &= \int_0^1 K(x,y)f(y)dy & (34)\\
&= \int_0^1 K(x,y)\left[g(y) + h(y)\right]dy & (35)\\
&= \int_0^1 K(x,y)g(y)dy + \int_0^1 K(x,y)h(y)dy & (36)
\end{aligned}$$

The left-hand integral represents the direct lighting term. It can be evaluated easily enough by sampling a point on a light source as described in the next section. The right-hand integral still has a recursive unknown term, $h(x)$, and hence we employ regular Monte Carlo integration, as before in order to evaluate this. It is important to note that these two integrals can be evaluated completely independently and therefore we can use different PDF's in each case and hence use different sample points y . In effect we partition the hemisphere of directions above x into two parts. One part is composed of all those directions which lead to light sources and the other is composed of those directions which do not: the former is evaluated using the left-hand integral, the latter is evaluated using the right (see figure 9). This partition is easily accomplished in the left-hand case by simply choosing sample points which are on light sources. For the indirect part we choose sample directions as before but *reject* those directions which hit light sources. We

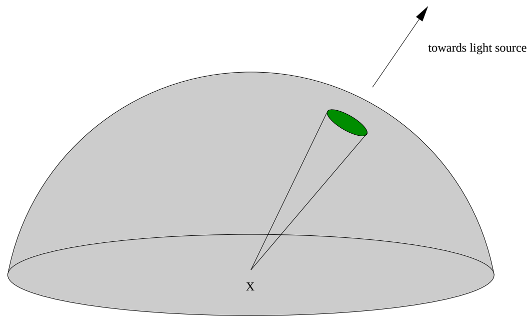


Figure 9: In this diagram we have one light source and the angle subtended by this light source is shown. The hemisphere above X is thus split into two partitions. The directions within the cone are sampled according to the direct lighting integral, those outside the cone are sampled according to the indirect lighting integral.

can now present pseudo-code which illustrates the basic path tracing with next event estimation algorithm.

```
ShadeRay(Ray,Intersection)
    If(Intersection not valid)
        return black
    Compute Surface Normal and Surface Properties
    Colour=Shade(Intersection)
    return(Colour)
End
Shade(Intersection)
    DirectColour=LightRay(Intersection)  // sample light source(s)
    If(reached maxdepth)
        return DirectColour
    NewDir = GetDiffuseDirection()    // using equation29
    RefColour=ReflectRay(NewDir)
    return(DirectColour+ (RefColour*(DiffusePhong/PDF)))  // this is k_d (see equation 30)
End
LightRay(Intersection)
    X=Random Point on LightSource(s)
    Colour=LightFrom(X)*DiffusePhong/PDF(X) // PDF is probability of choosing X
    return Colour
End
ReflectRay(Dir)
    Intersection=TraceRay(Dir)    // Find intersection point
    If(Intersection is a Light Source
        return(black)
    Else
        Colour=ShadeRay(Ray,Intersection)
        return(Colour)
```

```
End
```

5.2 Direct Lighting Calculations

5.2.1 Overview

The direct lighting calculation is at the core of most rendering systems. It calculates the reflected radiance from a point x in direction Θ_x as a result of incoming radiance coming directly from light sources. Recall the rendering equation.

$$L(x, \Theta_x) = L_e(x, \Theta_x) + \int_{\Omega_x} L(y, \Theta_y) f_r(x, \Theta_y, \Theta_x) \left|\Theta_y . N_x\right| d\omega_y \tag{37}$$

If we omit the surface radiance term (emitted radiance) and replace $L_f(x, \widehat{\omega}')$ with $L_e(x', \widehat{\omega}')$ which represents the emitted light from some point x' where x' is a point visible from x then we get:

$$L(x, \Theta_x) = \int_{\Omega} g(x, y) f_r(x, \Theta_y, \Theta_x) L_e(y, \Theta_y) \left|\Theta_y . N_x\right| d\omega_y \tag{38}$$

$g(x, x')$ is a visibility function which gives 0 if the path from x to y is occluded and 1 otherwise. This equation then embodies the direct lighting calculation.

Many rendering techniques make simplifications regarding the nature of light sources. The most common one is the use of point sources. These sources do not have surface area or volume but instead are comprised of single points that emit light equally in all directions. If we have a finite number N of point sources then the direct lighting calculation becomes:

$$L(x, \Theta_x) = \sum_{i=1}^{N} g(x, y_i) f_r(x, \Theta_{y_i}, \Theta_x) L_e(y_i, \Theta_{y_i}) \left|\Theta_y . N_x\right| \tag{39}$$

Ray tracing adopts this approach. The visibility term is evaluated by casting a ray, which is often called a shadow feeler, from x to y_i.

If however we wish to model light sources that have surface area and hence behave like real light sources we must evaluate incoming light from all points on the surface of the light source. Ray tracers tend to avoid this problem by modeling surfaces of light sources as discrete sets of point sources. For example, RayShade treats rectangular area light sources as rectangular grids of point sources. This is an approximation of the correct direct lighting.

Suppose however we wish to try and calculate this accurately. To do this we have to integrate over the surface points of the source. To find an equation for this we follow Kajiya and expressing the rendering equation as an integral over all surface points [10] (rather than an integral over all directions):

$$L(x, \Theta_x) = L_e(x, \Theta_x) + \int_{y} g(x, y) f_r(x, \Theta_y, \Theta_x) L(y, \Theta_y) \left|\Theta_y . N_x\right| \frac{\left|\Theta_y . - N_y\right|}{\left\|\overrightarrow{x - y}\right\|^2} dA(y) \tag{40}$$

N_x is the surface normal and N_y is the normal to the light source surface. Again by dropping the surface radiance term (emitted light) and replacing the reflected light term inside the integral with the emitted light term we get a formula for the direct lighting integral:

$$L_D(x, \Theta_x) = \int_y g(x, y) f_r(x, \Theta_y, \Theta_x) L_e(y, \Theta_y) \left|\Theta_y . N_x\right| \frac{\left|\Theta_y . - N_Y\right|}{\left\|\overrightarrow{x - y}\right\|^2} dA(y) \tag{41}$$

We can simplify this by making θ_1 the angle between N_x and the line from x to y, θ_2 the angle between N_y and the line from y to x, d the distance from x to y, and, assuming diffuse luminaires, I_s the intensity of the light source in any direction.. This gives us:

$$L_D(x, \Theta_x) = \int_y g(x, y) f_r(x, \Theta_y, \Theta_x) I_s \cos\theta_1 \frac{\cos\theta_2}{d^2} dA(y) \tag{42}$$

Since this is an integral equation over the surface area of the source we can use Monte Carlo integration to solve this by picking random sample points y on the surface and averaging the results. The visibility term can be evaluated by extending a ray from x to y and testing for intersections with other objects or light sources. The BRDF can be evaluated by using a number of possible lighting models and the angular and distance terms are simple functions of the geometry.

As we have seen already the simplest lighting model to use as a BRDF is the basic diffuse reflection model. This means that the diffuse reflection in any direction is calculated by dividing the diffuse reflection coefficient of the surface, k_d, by π. So, our equation becomes:

$$L_D(x, \Theta_x) = \int_y g(x, y) k_d I_s \cos\theta_1 \frac{\cos\theta_2}{d^2 \pi} dA(y) \tag{43}$$

Basic Monte Carlo with no importance sampling with N samples will give us:

$$L_D(x, \Theta_x) \approx \frac{I_s k_d}{\pi N} \sum_{i=1}^{N} g(x, y_i) \cos\theta_1 \frac{\cos\theta_2}{d^2} \tag{44}$$

If we sample according to some probability density function (PDF) then our equation becomes:

$$L_D(x, \Theta_x) \approx \frac{I_s k_d}{\pi N} \sum_{i=1}^{N} g(x, y_i) \cos\theta_1 \frac{\cos\theta_2}{p(y) d^2} \tag{45}$$

where $p(y)$ is the probability of choosing y. We now move on to discuss sampling strategies for spherical and triangular light sources. This work closely follows that of Shirley [18].

5.2.2 Sampling Spherical Light Sources

Sampling a particular type of light source involves choosing a PDF for the surface of the light source and picking sample points according to this PDF. The simplest form of PDF for a spherical light source is a uniform distribution on the surface of the sphere. This gives us the so-called p_1 estimator. The density function is simply:

$$p_1(y) = \frac{1}{4\pi r^2} \tag{46}$$

Putting this into Equation 45 yields:

$$L_D(x, \Theta_x) \approx \frac{4r^2 I_s k_d}{N} \sum_{i=1}^{N} g(x, y_i) \cos\theta_1 \frac{\cos\theta_2}{d^2} \tag{47}$$

In order to get uniform sample points on the sphere we generate two random numbers $\varepsilon_1$and ε_2a nd using a polar coordinate system with ϑ being the polar angle, and ϕ being the azimuthal angle we have:

$$(\vartheta, \phi) = (\arccos(1 - 2\varepsilon_1), 2\pi\varepsilon_2) \tag{48}$$

We then convert this into Cartesian coordinates to get a sample point on the sphere:

$$y = \begin{bmatrix} cx + 2r\cos(2\pi\varepsilon_2)\sqrt{\varepsilon_1(1-\varepsilon_1)} \\ cy + 2r\sin(2\pi\varepsilon_2)\sqrt{\varepsilon_1(1-\varepsilon_1)} \\ cz + r(1 - 2\varepsilon_1) \end{bmatrix} \tag{49}$$

where (cx, cy, cz) is the centre of the light source and r is its radius.

The p_1 estimator has been implemented and incorporated into the RayShade system. In order to do this properly we must employ rejection sampling which will discard sample points chosen which are on the opposite side of the sphere. The set of images shown in Figure 10were all produced using the p_1 estimator with a larger value for N (the number of samples) per light source as we go from left to right. Note the smooth shading on the back wall.

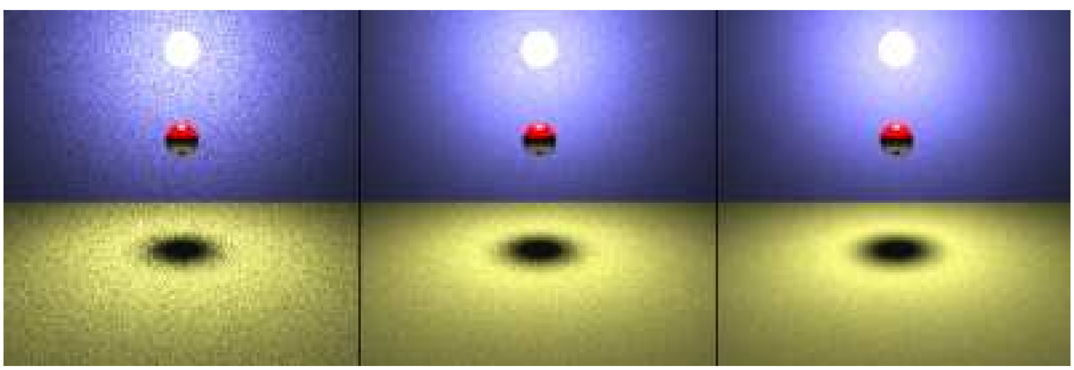


Figure 10: The p1 estimator: from left to right, N=1,5, and 25

As we can see from the images above, there is a high variance associated with the use of the estimator, even at relatively large numbers of samples per light source. This can be improved dramatically by sampling uniformly in directional space, or using what Shirley calls the p_2 estimator. This is equivalent to sampling uniformly on the space of directions subtended by the light source. This is shown in figure **??**. In other words, sampling from the space of directions towards the sphere as seen from x. If we define a right-handed orthonormal basis $\widehat{u}\widehat{v}\widehat{w}$ where $\widehat{w}$ is coincident with $\overrightarrow{xc}$, and $\widehat{v}$ is found by the normalised cross product of $\widehat{n}$ and $\widehat{w}$, then the direction to x' can be calculated in terms of polar and azimuthal angles to the $\widehat{u}\widehat{v}\widehat{w}$ coordinate system as:

$$\begin{bmatrix} \vartheta' \\ \phi' \end{bmatrix} = \begin{bmatrix} a\cos(1 - \varepsilon_1 + \varepsilon_2\sqrt{1 - \left(\frac{r}{\|x-c\|}\right)^2} \\ 2\pi\varepsilon_2 \end{bmatrix} \tag{50}$$

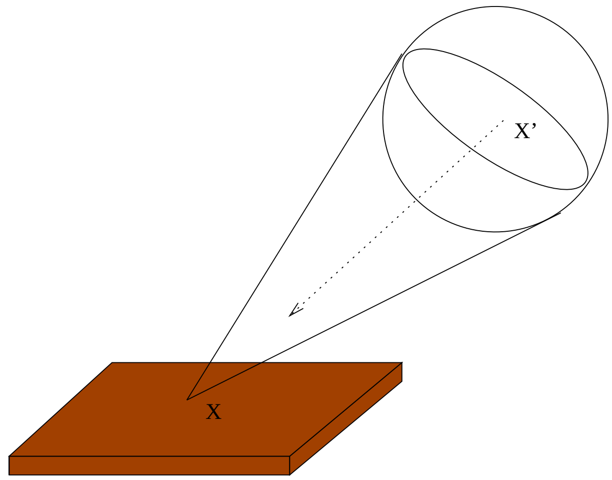


Figure 11: Directional space spubtended by a spherical light source. X represents the point we are shading and X' is the random point chosen within the solid angle.

This gives us the direction to y. We transform this to Cartesian coordinates and calculate the intersection with the sphere in order to get y. We also need an equation of the PDF with respect to the area of the sphere. This is called the p_2 estimator and can be expressed as:

$$p_2(y) = \frac{\cos\theta_{max}}{2\pi d^2 \left(1 - \sqrt{1 - \left(\frac{r}{\|x-c\|}\right)^2}\right)} \qquad (51)$$

where $\cos\theta_{max}$ is the cosine of the angle between the vector from x to y and the normal to the sphere at y. We have implemented this estimator into our system also and the images in Figure 12 show the results of using it on the same scene as before.

Figure 12: The p2 estimator: from left to right, N =1,5, and 25

As we can see the images produced by the p_2 estimator are far smoother than those produced by the p_1. Also, increasing the number of samples per light source does not improve the quality in terms of the direct shading. In fact the shading produced using just one sample per light source is perfectly adequate. However, increasing N does improve the shadowing on the ground. In practice this type of direct lighting will be used in the context of a path tracer which will send numerous rays per pixel which will mitigate the need to send more than one shadow ray per light source anyway. It should also be noted that the p_2 estimator is far more efficient because we don't discard samples.

Shirley also describes a p_3 estimator. This estimator samples the cosine-weighted solid angle rather than sampling uniformly within the solid angle like the p_2. This produces better results but is more computationally expensive because it is necessary to numerically invert one of the formulae involved and hence does not tend to be used in practice.

5.2.3 Sampling Triangular Light Sources

This is not as simple to carry out as the spherical case because it is difficult to compute solid angles subtended by triangles. So instead we approximate the solid angle of a triangular source by choosing random points within a triangle which is formed by the projection of the triangular source onto the unit sphere centred at x. Figure 13illustrates this idea. The coordinates of the

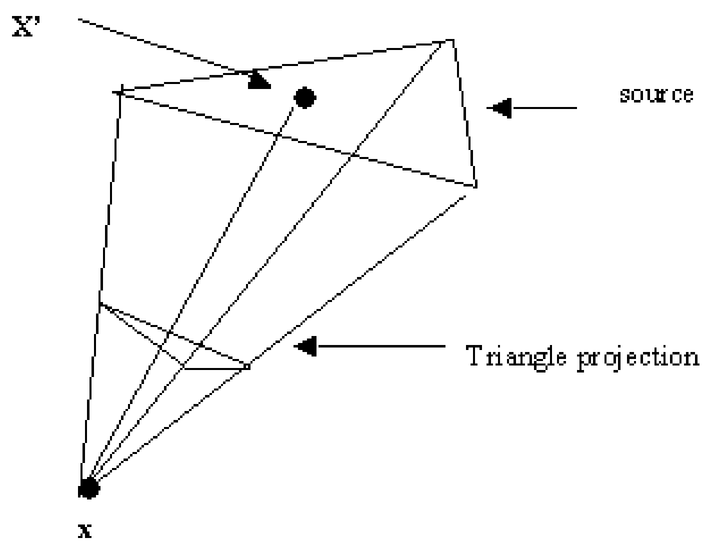


Figure 13: Sampling triangular light sources using projections

corners of the projected triangle,p'_1, p'_2, p'_3 , are:

$$p'_i = x + \frac{p_i - x}{dist(x, p_i)} \tag{52}$$

where p_1, p_2, p_3 are the coordinates of the triangular source. A random point on the projected triangle in coordinates is:

$$(u, v) = (1 - \sqrt{1 - \varepsilon_1}, \varepsilon_2 \sqrt{1 - \varepsilon_1} \tag{53}$$

In terms of Cartesian coordinates this is:

$$x' = p'_1 + u(p'_2 - p'_1) + v(p'_3 - p'_1) \tag{54}$$

We can then find y by finding the intersection of the ray in the direction of $\overrightarrow{xx'}$ with the triangular source. It remains to compute $p(y)$ which is given by:

$$p(y) = \frac{dist^2(x, x'') \cos\vartheta'}{area_{p'_1 p'_2 p'_3} dist^2(x, y) cos\vartheta''} \tag{55}$$

ϑ' is the angle between the source normal and the vector $\overrightarrow{yx}$ and ϑ" is the angle between $\overrightarrow{x''y}$ and the normal to the projected triangle.

We have implemented this sampling scheme in our system also and the results can be seen in Figure 14. This image has one triangular light source in an enclosed space. Note that the source only propagates light in one direction. The image was computed with one sample per direct lighting computation.

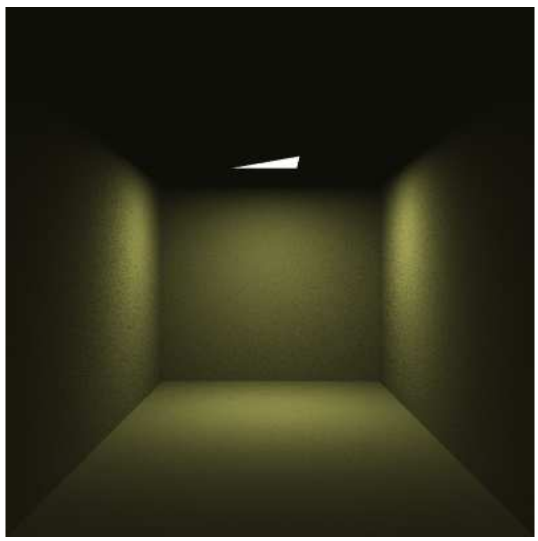


Figure 14: Direct lighting with triangular light source

5.3 Importance Sampling

The next step is to include a full specular/diffuse BRDF into our system. We can do this relatively easily by using the Phong model but this gives rise to two principal problems. Firstly, we are sampling ray directions by carrying out a form of importance sampling which concentrates on directions which are near to the normal. This is perfectly appropriate when we are dealing with diffuse reflection since the bulk of the significant light transport will be in these directions. However it is not so appropriate when we include specular reflection as, in this case, we need to concentrate on directions which are nearer to the specular reflection angle. Secondly the Phong model lacks certain properties which are necessary to ensure physical plausibility. Lafortune et al. [13] dealt with this problem by introducing a modified Phong reflectance model which is suitable for physically based rendering. We will start by presenting this model and then go on to describe how to sample specular directions effectively.

The physical properties which a physically plausible reflectance model must have are as follows:

- It should be reciprocal or in other words the incoming and outgoing light directions are interchangeable:

$$f_r(x, \theta_i, \theta_o) = f_r(x, \theta_o, \theta_i) \tag{56}$$

- It should ensure *conservation of energy* meaning that the fraction of incoming light from any direction reflected over the hemisphere is always less than 1:

$$\int_{\Omega_x} f_r(x, \theta_i, \theta_o) \cos\theta_o d\omega_o \leq 1, \perp \forall x, \theta_i \tag{57}$$

Lafortune also presents some desired practical properties which are necessary for a reflectance model:

- We must be able to evaluate the BRDF for any point on the surface and any incoming and outgoing direction.
- We need to be able to sample outgoing directions according to this BRDF. In other words we should be able to construct a probability density function which will closely follow the BRDF and hence sample important directions. We have been doing this already using the simple diffuse BRDF.
- It is sometimes desirable to be able to integrate the BRDF over the hemisphere.

The modified Phong reflectance model can be expressed as:

$$f_r(x, \theta_i, \theta_o) = \frac{k_d}{\pi} + k_s \frac{n+2}{2\pi} \cos^n \alpha \tag{58}$$

where α is the angle between the perfect specular reflection direction and the outgoing direction, k_d is the diffuse reflectivity as before, k_s is the specular reflectivity, and n is the specular exponent (see figure 15). We can show that this model satisfies the reciprocity principle and also that

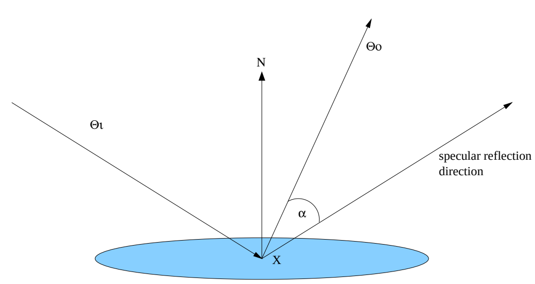


Figure 15: The modified Phong reflectance model

conservation of energy is guaranteed as long as $k_d + k_s \leq 1$. So, it satisfies the necessary physical properties. If we look at the desired practical properties we see that it easily satisfies the first one. In the case of the third one, we can't integrate analytically but Lafortune describes a Monte Carlo integration technique which is appropriate. The second property is crucial as the ability to choose sample directions according to the BRDF allows us to carry out importance sampling effectively. It turns out that we can't use the BRDF directly as a PDF. We can try to carry out some form of rejection sampling but this will be very inefficient. The normal practice therefore is to split the BRDF into a diffuse part and a specular part and to sample each of them separately.

This means that when we want to choose a ray direction we either randomly generate a diffuse ray (sampling the diffuse part of the BRDF) or randomly generate a specular ray (sampling the specular part). If we generate a diffuse ray then we multiply the result by the diffuse part of the Phong formula (and divide by the PDF of generating the ray) and if we generate a specular ray we multiply the result by the specular part of the Phong formula (dividing by the appropriate PDF as before)[4]. How do we choose whether we cast diffuse or specular rays? The obvious approach is to cast more diffuse rays from surfaces which are primarily diffuse and more specular rays from surfaces where the opposite is the case. So we associate with each surface a number p_d which represents the probability of casting a diffuse ray and a number p_s which represents the probability of casting a specular ray. [5]We then generate a random number ε such that $0 \leq \varepsilon \leq 1$ and cast a diffuse ray if $0 \leq \varepsilon \leq p_d$ and cast a specular ray otherwise. The estimator must be accordingly divided by p_d or p_s, according to which type of ray was cast, in order to weight the samples correctly.

Casting a diffuse ray can be carried out in the same manner as described in section 4.2. To cast a specular ray we use the following PDF:

$$pdf(\theta_i) = \frac{n+1}{2\pi} \cos^n \alpha \tag{59}$$

This can be sampled by choosing two random number $\varepsilon_1$and ε_2 and calculating the direction as:

$$(\alpha, \phi) = (\arccos \varepsilon_1^{\frac{1}{n+1}}, 2\pi\varepsilon_2) \tag{60}$$

In vector coordinates this is:

$$(x, y, z) = \left(\sqrt{1 - \varepsilon_1^{\frac{2}{n+1}}} \cos(2\pi\varepsilon_2), \sqrt{1 - \varepsilon_1^{\frac{2}{n+1}}} \sin(2\pi\varepsilon_2), \varepsilon_1^{\frac{1}{n+1}}\right) \tag{61}$$

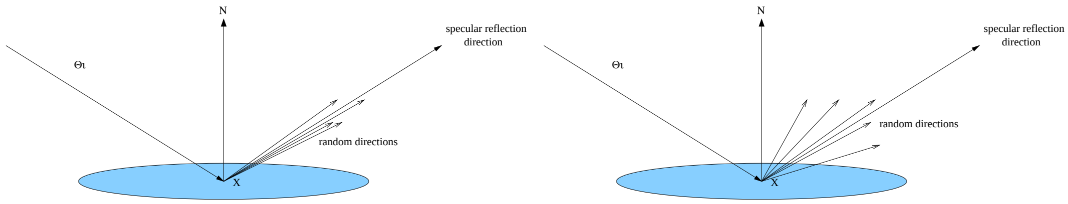
In this way we can ensure that we sample directions which are close to the specular reflection angle. Note that the PDF ensures that the specular lobe is sampled properly. In other words if we have a very high value of n (the specular exponent) associated with the surface then the range of random directions chosen will be tightly grouped around the perfect specular reflection directions (see figure 16). In fact, as n tends towards infinity, the path tracer will behave as a normal ray tracer, shooting rays along the specular reflection angle. This means that for highly specular surfaces we get low variance and hence smooth pictures.

5.4 Implementation and Results

We need to find an expression for the radiance along a path using next event estimation as we did for pure path tracing (see equation 24). We use the notation introduced in equation 43 whereby

[4]This discussion assumes that we are not employing Russian Roulette. In order to produce an unbiased solution we should do so however the removal of the resulting small amount of bias comes at the expense of the addition of a small amount of noise. Lafortune [13] provides the details as to how to add Russian Roulette to the algorithm.

[5]If we are using an RGB system the most appropriate means of doing this is to make p_d the average of the three diffuse reflection coefficients and p_s the average of the three specular reflection coefficients. These should be normalised in some manner so that $0 \leq p_d \leq 1$, $0 \leq p_s \leq 1$, and $p_d + p_s = 1$.



(a) A high value of n means directions are chosen which are close to the specular reflection angle

(b) A lower value of n results in a wider range of directions being sampled

Figure 16: Specular lobe importance sampling

$L_D(x, \Theta_x)$ represents the radiance from a point x in direction Θ_x due to direct lighting (see section 5.2.1. for details as to how to calculate this). We will also define a transport operator T_x which represents, for any point x on the path, the term by which we multiply the indirect lighting. More formally:

$$T_{xi} = \frac{f_r(x_i, \Theta_i, \Theta_{i-1}) \cos \Theta_{xi}}{pdf(\Theta_{xi}).p_{dir}} \tag{62}$$

where p_{dir} is p_d if a diffuse ray was cast from this point and p_s otherwise. Figure 17 shows the calculations which are necessary for the case of paths of length 2. Similarly paths of length 3 are

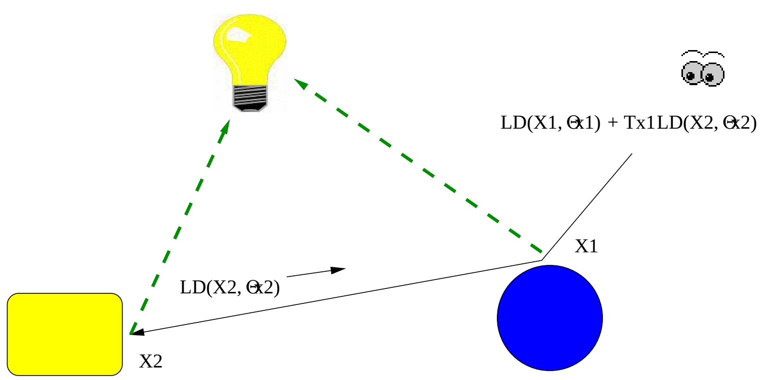


Figure 17: Calculations for paths of length 2

calculated as:

$$L_{PTNEE}(x0, \Theta_{x0}) = L_D(x1, \Theta_{x1}) + T_{x1}\left(L_D(x2, \Theta_{x2}) + T_{x2}\left(L_D(x3, \Theta_{x3})\right)\right) \tag{63}$$

We can generalise this to paths of length n as follows:

$$L_{PTNEE}(x0, \Theta_{x0}) = \sum_{i=1}^{n} L_D(xi, \Theta_{xi}) \left(\prod_{j=0}^{i-1} T_{xi} \right) \tag{64}$$

where T_{x0} is 1.

The implementation PTNEE employs both next event estimation and importance sampling using the modified Phong BRDF. The pseudo-code for the main loop of the system is shown below.

```
ShadeRay(Ray,Intersection)
    If(Intersection not valid)
        return black
    Compute Surface Normal and Surface Properties
    Colour=Shade(Intersection)
    return(Colour)
End
Shade(Intersection)
    DirectColour=LightRay(Intersection)  // sample light source(s)
    If(reached maxdepth)
        return DirectColour
    r = random number                 // between 0 and 1
    if (r < pd)
        NewDir = GetDiffuseDirection()
        RefColour=ReflectRay(NewDir)
        return(DirectColour + (RefColour*(DiffusePhong/PDF))
    else
        NewDir = GetSpecularDirection()
        RefColour=ReflectRay(NewDir)
        return(DirectColour+ (RefColour*(SpecularPhong/PDF)))
End
LightRay(Intersection)
    X=Random Point on LightSource(s)
    Colour=LightFrom(X)*FullModifiedPhong/PDF(X)
    return Colour
End
ReflectRay(Dir)
    Intersection=TraceRay(Dir)    // Find intersection point
    If(Intersection is a Light Source
        return(black)
    Else
        Colour=ShadeRay(Ray,Intersection)
        return(Colour)
End
```

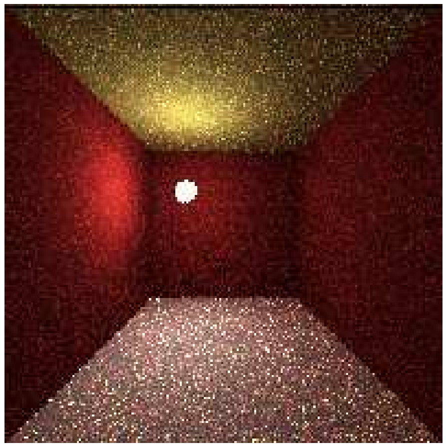
We have tested this algorithm using the same scene as used in figure 8. Since all the surfaces in this scene are completely diffuse the new system will carry out lighting calculations and choose ray directions in exactly the same manner as the previous one. The difference between the two therefore lies entirely in the fact that the new version employs next event estimation. The results of this can be seen in figure 18. If we compare these to the results obtained for pure path tracing two things are immediately apparent. Firstly, the images are vastly improved in quality. The results obtained using 9 samples per pixel is noticeably smoother than that obtained using 1024 samples per pixel in the previous case. Secondly we should note that this has been achieved at the expense of some extra computation time. The PTNEE images take of the order of 3 times as much CPU time to compute as the PT ones. The reason for this is the extra burden imposed by the necessity to cast shadow rays. However, the advantage of smoother images hugely outweighs the disadvantage incurred by longer running times. We can see that at 64 samples per pixel we are getting excellent results and at 256 samples the results are near perfect.

The graph displayed in figure 19 shows the results of convergence measurements using the version computed at 1024 samples per pixel using PTNEE as a reference image. We graph the results obtained from PT and from PTNEE. As we can see from the graph, when using PT, we need to increase the number of samples per pixel into the region of 1024 before we get results approaching those of PTNEE with just 1 sample per pixel.[6]. . This gives an indication of how quickly the images converge to the correct solution using the two different methods and using different numbers of samples per pixel.

The next sequence of images show the results of using the system on a scene which contains specular surfaces. The scene is similar to the previous one except the red walls at the back and sides of the room now have a strong specular component and we have added a green sphere floating in the middle of the room (this is also specular). Figure 20 shows the results which we obtain when we do not use importance sampling of the specular lobe. In this case we choose ray directions using the cosine distribution and multiply the results by the full modified Phong BRDF (as shown in equation 58). Figure 21 however shows the improvement which results from using specular lobe importance sampling as described in section 5.3. For the same number of samples that were used to create the images in figure 20 we obtain far superior results.

Finally as an complete example of rendering a specular scene, figure 22 shows the same scene as above rendered at 1024 samples per pixel. Note the specular highlights, the reflections in the specular surfaces, and the colour bleeding which is present.

[6]The convergence measurements were computed as follows. The PTNEE version at 1024 samples per pixel was used as a reference image and each of the other images were 'subtracted' from this one on a pixel by pixel basis. The results of this were added together to form a numeric measure. So the discrepancy between an image and itself would be 0. All images are 200x200 so an image where each pixel differs in it's red, green and blue values by 1, from the reference will return a discrepancy of 120,000.



(a) 1 sample, 13 secs

(b) 9 samples, 1 min 55 secs

(c) 25 samples, 5 mins 29 secs

(d) 64 samples, 13 mins 49 secs

(e) 256 samples, 54 mins 49 secs

(f) 1024 samples, 3 hrs, 39 mins, 37 secs

Figure 18: Images computed with next event estimation

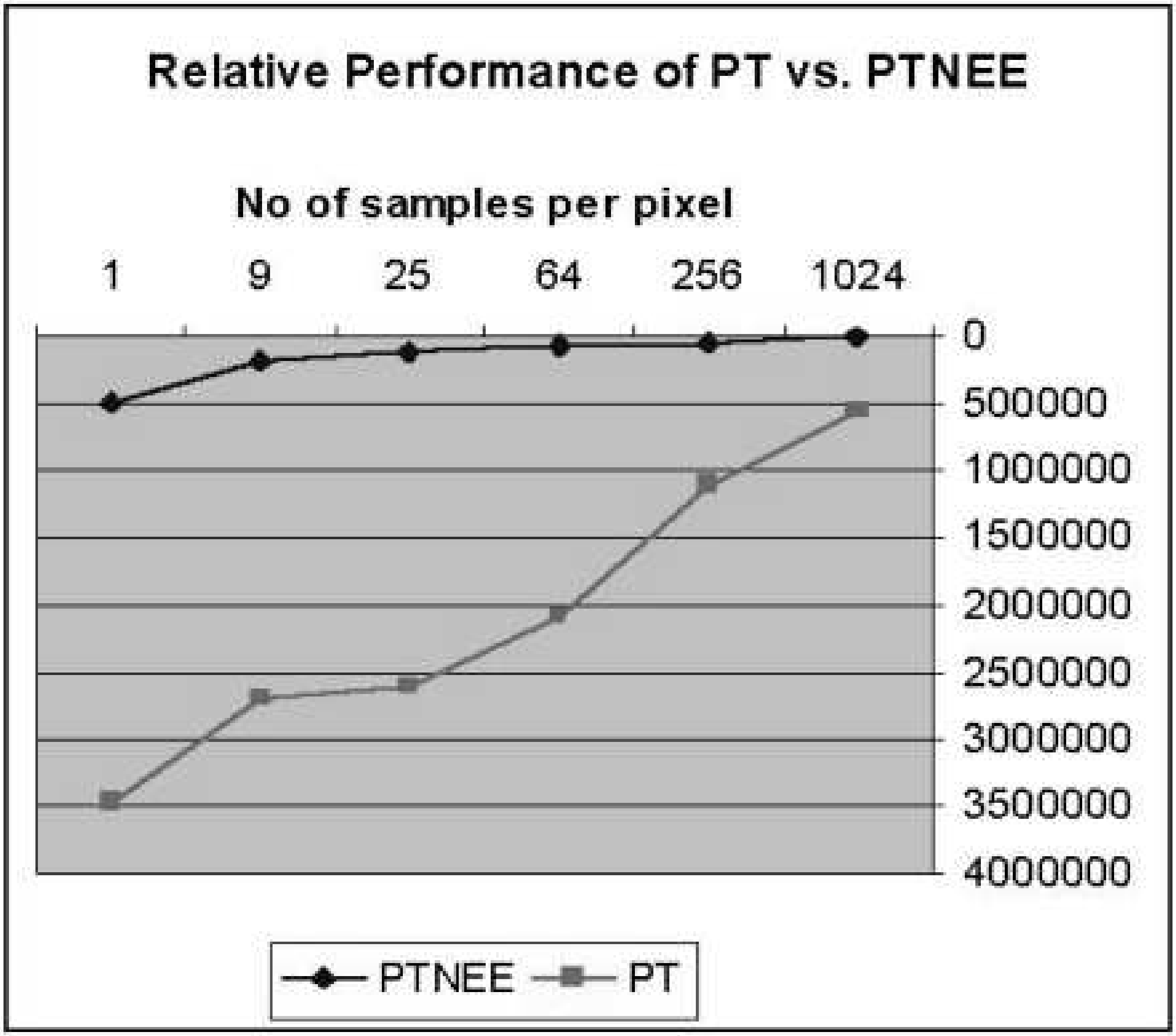


Figure 19: Convergence testing

(a) 9 samples (b) 25 samples (c) 64 samples

Figure 20: Images computed without importance sampling

(a) 9 samples (b) 25 samples (c) 64 samples

Figure 21: Images computed with importance sampling

Figure 22: Path tracing with next event estimation and importance sampling.

6 Bidirectional Path Tracing

This section describes a more generalised form of path tracing which, as well as constructing paths from the eye, also constructs paths from the light source. It then connects these paths and estimates the flux which travels along them. Rendering by constructing light paths is a well-known technique (for example see [3],[15]). It is superior to path tracing in some cases involving indirect lighting. Bidirectional path tracing combines both techniques in order to produce a method which combines the advantages of both. We start by describing the types of scenes which this technique is designed to handle, before moving on to present the mathematical and algorithmic details. We conclude by describing our implementation and the results which we obtained. Bidirectional path tracing was independently introduced by Veach [21] and Lafortune [12]. Their methods of combining paths differ in some respects. We will finish by discussing these variations.

6.1 Dealing with Difficult Lighting Conditions

The lighting conditions which path tracing finds difficult are those which involve scenes which are lit primarily indirectly. It is not difficult to see from an intuitive point of view why this is the case. Scenes which are lit indirectly are ones where the bulk of the illumination present on surfaces has been reflected off another surface already. For example consider the scene shown in figure 23.

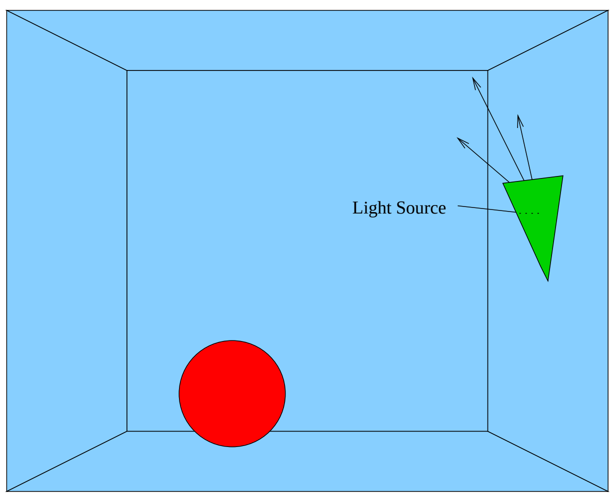


Figure 23: In this case the light source is inside the cone on the right wall. The only areas of the scene which are directly illuminated are parts of the right wall and the ceiling.

A path tracer will extend rays into the scene and those rays which hit portions of the scene which are indirectly illuminated will contribute nothing initially. This is because the direct lighting shadow rays will return nothing since the light source is occluded. At the next bounce we will more than likely get nothing also, unless the second point on the path happens to land within the indirectly light area. Those rays which do happen to land in this area will return large radiance values and the result will be noisy images. As an example of this consider the image in figure 24. This scene is lit by 6 spherical light sources each of which is enclosed within a cone and therefore occluded with respect to most of the scene. The only areas of direct lighting are above the cones on the left and right walls and to some extent on the ceiling. Path tracing clearly fails

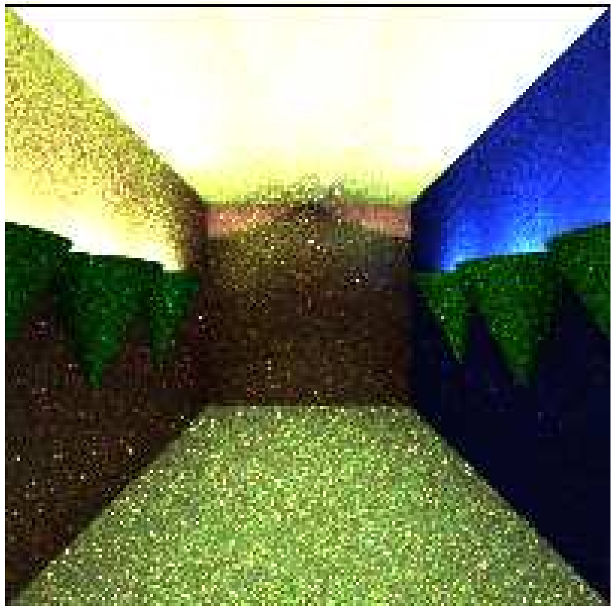


Figure 24: Image computed using path tracing at 1024 samples per pixel.

to re-construct the indirect lighting on the floor, even with large numbers of samples per pixel. The reader should note that in the real world occluded light sources tend to be the rule rather than the exception. Most light sources are enclosed in shades of some description so in effect most scenes (or at least most indoor scenes) are primarily lit by indirect lighting.

The light tracing techniques of [3] and [15] are designed to overcome these problems and do so effectively. However these techniques suffer from some related difficulties. Light paths which are traced from the light source will only contribute radiance values to the final image if they happen to pass through the image plane and reach the viewpoint (as in figure **??**). Much of the time this will not happen and hence we need to send huge amounts of light paths in order to construct images. This is the same problem as we encountered with pure path tracing, but in reverse. Once again we can counter this however by carrying out a form of next event estimation. In this case instead of sampling the light source at each step of the path we sample the viewpoint at each step of the path. This leads to a feasible algorithm. However once again there are certain classes of scenes which will prove difficult for this technique. These situations are ones whereby the bulk of the scene is not visible from the viewpoint from which we are rendering. Figure 25 shows this situation.

In this case most of the scene is not visible. When light paths are constructed they will only contribute if they happen to land within the area indicated (in the same way that eye paths only contribute in the path tracing case if they land within the area of direct illumination). Note that this limitation of light tracing applies in cases where the scene is directly lit as well. So, the above case will present severe difficulties for both techniques. These types of problems provided the motivation for the development of bidirectional path tracing.

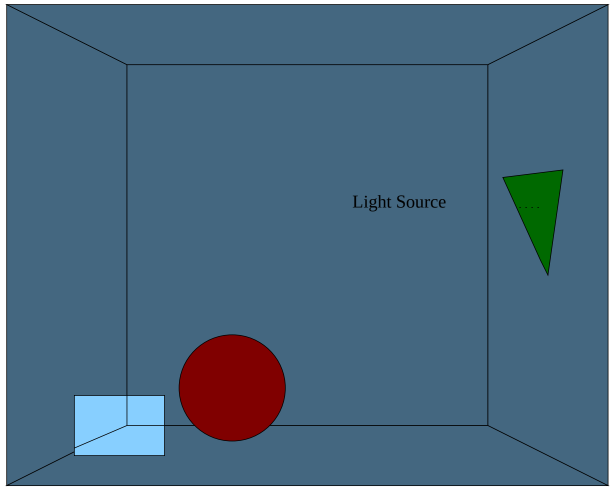


Figure 25: In the diagram above the rectangle in the lower left represents the portion of the scene which is visible from the current viewpoint.

6.2 Combining Eye Paths and Light Paths

The basic idea behind bidirectional path tracing is relatively simple. Rather than simply constructing paths from the eye, or constructing paths from the light source, we construct paths from both directions simultaneously, connect these paths together and then compute the light transport along these combined paths. In this way we expect that even in situations where scenes are primarily lit by indirect lighting and not directly visible to the viewer we will still be able to find feasible light transport paths. Figure 26 shows this idea in practice.

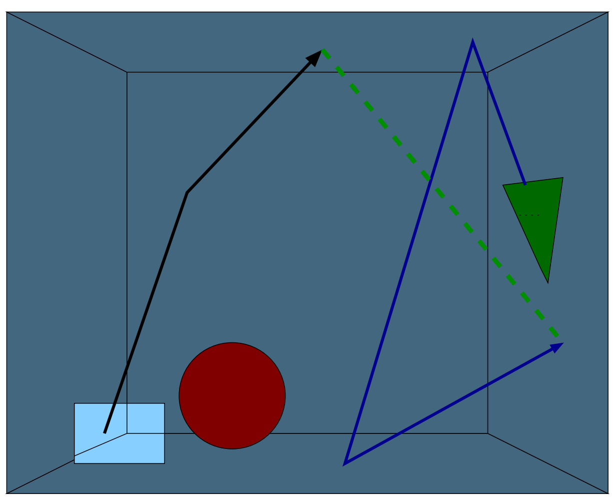


Figure 26: The black line represents and eye path and the blue line represents a light path. The dotted line connects the two of them and provides us with a complete light transport path from eye to light.

The paths are constructed in the usual manner (e.g. each new ray direction is determined probabilistically according to some suitable PDF), however the final step which connects the last point on the eye path with the last point on the light path is a deterministic step, and is equivalent to casting a shadow ray to test for occlusion. Of course it should be immediately obvious that we do not have to restrict ourselves to merely connecting the end-points on the paths. There is no reason why we should not connect all points on the eye path with all points on the light path (testing for occlusion each time) and hence produce a whole set of light transport paths from the initial two paths.

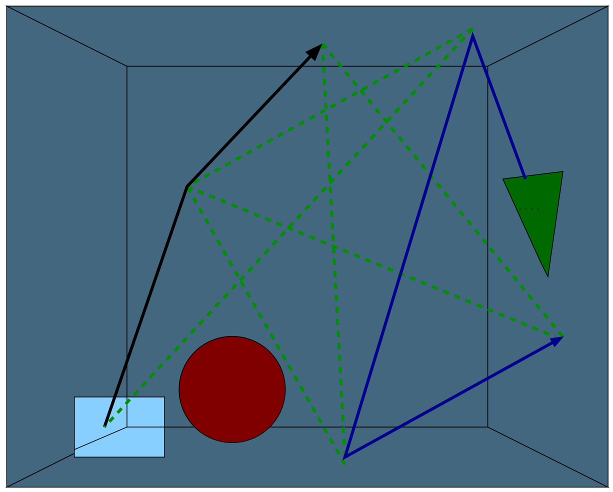


Figure 27: All points on both paths connected.

Figure 27 shows this situation. Note that we don't connect all points on both paths. In several cases pairs of points are occluded and in other cases these pairs of points lie on the same surface, in which case we don't connect them. In the case of figure 27 we get six non-occluded light transport paths. We shall refer to these constructed paths as *sub-paths*. How many possible sub-paths do we get from a given number of eye and light steps? Suppose we take m eye steps and n light steps as shown in figure 28.

Note that we are regarding an eye step as being a ray traced into the scene from the eye and similarly a light step as a ray traced into the scene from a light source. This is an intuitive convention but somewhat different from the convention offered by Veach [21] for example. Veach regards choosing a point on the pixel and choosing a point on the light source as constituting steps in themselves. This allows him to construct a more generic formulation which admits the possibility of including pure path tracing and pure light tracing (i.e. the equivalent of pure path tracing in reverse) into his framework. This is useful primarily from a theoretical point of view as these techniques do not constitute practical rendering algorithms.

We can easily calculate that the number of possible sub-paths, s, is as follows. An eye-path of length m will consist of $m+1$ points (including the point on the pixel) and similarly a light-path of length n will consist of $n+1$ points. Any of these points can be connected using a deterministic step[7] and hence we have the following number of possible sub-paths, s:

$$s = (m+1)(n+1) \tag{65}$$

[7]The deterministic step always connects points between the eye path and light path. It would also be possible to use a deterministic step to connect previuously unconnected points on the eye path (or indeed two points on the light path) and hence generate a new path in this manner. In general this is not done however.

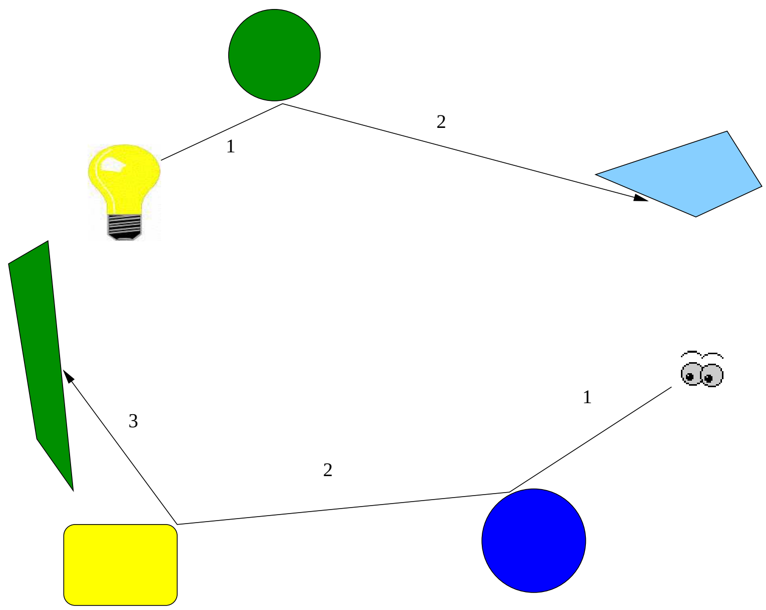


Figure 28: m=3 eye steps and n=2 light steps

Some of these paths essentially constitute next event estimation steps. Consider the paths where n is 0, i.e. we are connecting points on the eye path directly with the first point on the light path (see figure 29). We are essentially carrying out direct lighting in this case and in fact bidirectional path tracing with no light steps is equivalent to the path tracing with next event estimation as outlined in section 5. [8] What about the case where we don't take any eye steps? This situation is shown in figure 30. This is equivalent to direct sampling of the viewpoint (as would normally be carried out when doing light tracing). It forms a useful estimator if the sub-path passes through a pixel on the image plane before it reaches the viewpoint. In our implementation we have not included a facility to handle these paths as doing so would require significant rearrangement of the basic ray tracer upon which it is built[9]. Therefore the number of sub-paths, s, which we have is:

$$s = (m)(n + 1) \tag{66}$$

Figure 31 shows all of these sub-paths. Two main issues remain to be discussed:

- How do we calculate the radiance along one of these sub-paths?
- How do we combine the radiance from various sub-paths together in order to form an estimator for the radiance through the pixel?

The first question is answered simply enough as similar calculations are involved as in the path tracing case. The main difference is that there is a deterministic step in the middle. In the next

[8] Strictly speaking this is equivalent to path tracing with next event estimation using Shirley's p1 estimator to do the direct lighting since we are choosing a completely random point on the light source to sample rather than sampling within the solid angle as done by the p2 estimator. In practice it is better when implementing bidirectional path tracing to actually use the p2 estimator instead.

[9] This is because the system works on a pixel by pixel basis. To incorporate direct sampling of the viewpoint we would have to build in the capability of storing radiance contributions to pixels other than the current pixel being computed.

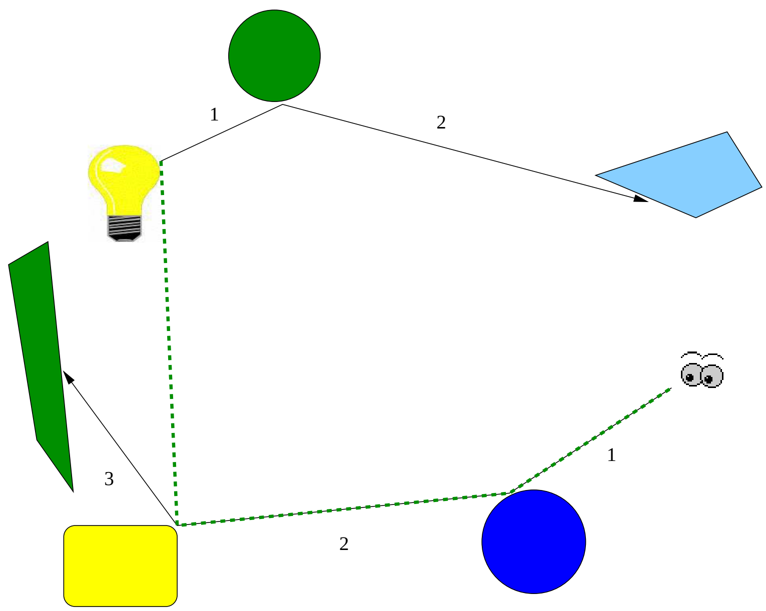


Figure 29: The sub-path is shown in green and is formed by connecting the third point on the eye path with the first point on the light path.

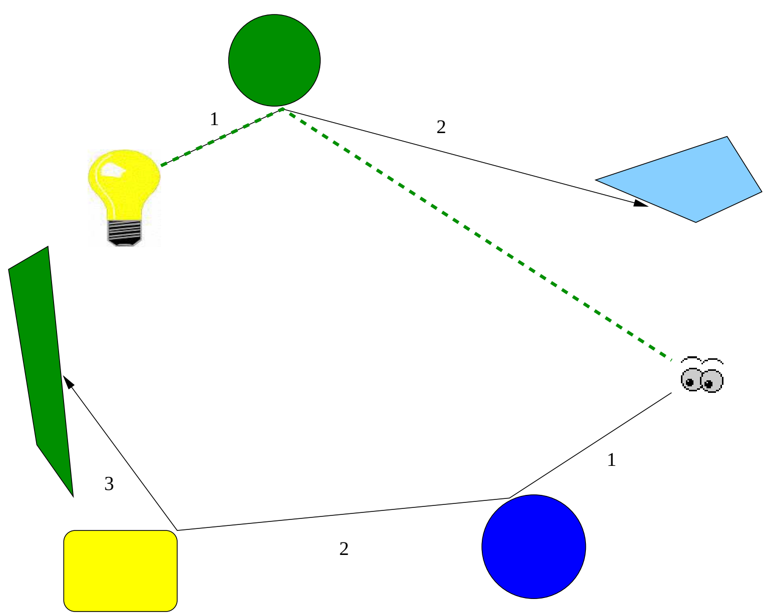


Figure 30: The sub-path is shown in green and is formed by connecting the second point on the light path with the first point on the eye path.

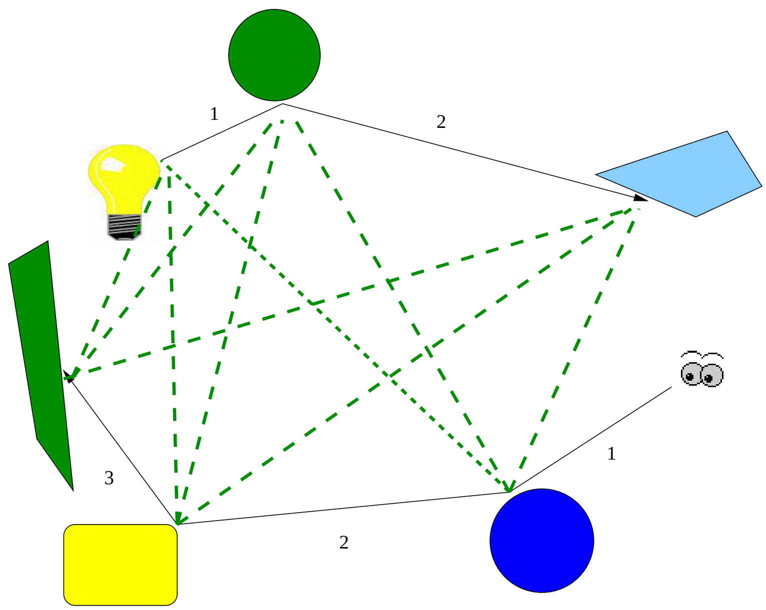


Figure 31: All possible sub-paths

section we will present the mathematical and algorithmic details as to how to go about doing this. The second question is somewhat more subtle. It turns out that we have tremendous freedom as to how to we combine the various primary estimators at our disposal. The important point to understand is that the length of the sub-path is the crucial factor. In regular path tracing we might send N number of paths per pixel. Suppose we are not using Russian Roulette and therefore all paths are of length L (where $L > 1$). Each such path provides us with a means of estimating the radiance which has left the light source(s) and travelled to the viewpoint by means of 1 bounce, by means of 2 bounces, 3 bounces and so on (for numbers of bounces less than or equal to L). So we effectively have N estimators for the direct lighting (1 bounce), N estimators for the indirect lighting (2 bounces), N estimators for the indirect-indirect lighting (3 bounces) and so on. How do we combine these N estimators to form a single secondary estimator for the radiance resulting from l bounces? We simply add up all the contributions for each path to get a value for a single path, add the values for all the paths together, and divide by N. We would obtain exactly the same result if we stored all of the results for 1 bounce, added them all together, and then divided by N; did the same for two bounces, and three and so on; and then added them all together to get an estimate for the pixel.[10]We can think of the problem of combining different estimators from a bidirectional path in the same way. For example, given a single path consisting of two eye steps and two light steps, we have two different estimators for the radiance contributed by paths of length four (the second point on the eye-path connected with the third point on the light-path, and the third point on the eye-path connected with the second point on the light path). If we have l different estimators then the simplest approach is to simply evaluate all l estimators, add the results together, and divide by l. Effectively what we are doing in this case is assigning uniform weights (of $\frac{1}{l}$) to each of the estimators. However we can assign any weights that we like as long as their sum (for a given path length) is 1.

[10]We don't do this because of the resulting large memory requirements

This does not however give us the best results. Lafortune [12] describes how to weights the paths so that the more important ones are more likely to get higher values and Veach [22] describes how to adapt the algorithm so that an optimal combination of estimators can be achieved. We will look at these issues in section 6.4 but for the moment assume uniform weighting.

6.3 Implementation and Results

So how do we go about calculating the radiance along one of these sub-paths? Suppose we have a sub-path of consisting of l steps which we denote as $x_0, x_1, x_2, ..., x_l$. Suppose also that this sub-path has been constructed using m eye steps and n light steps as in figure 32. Then we have

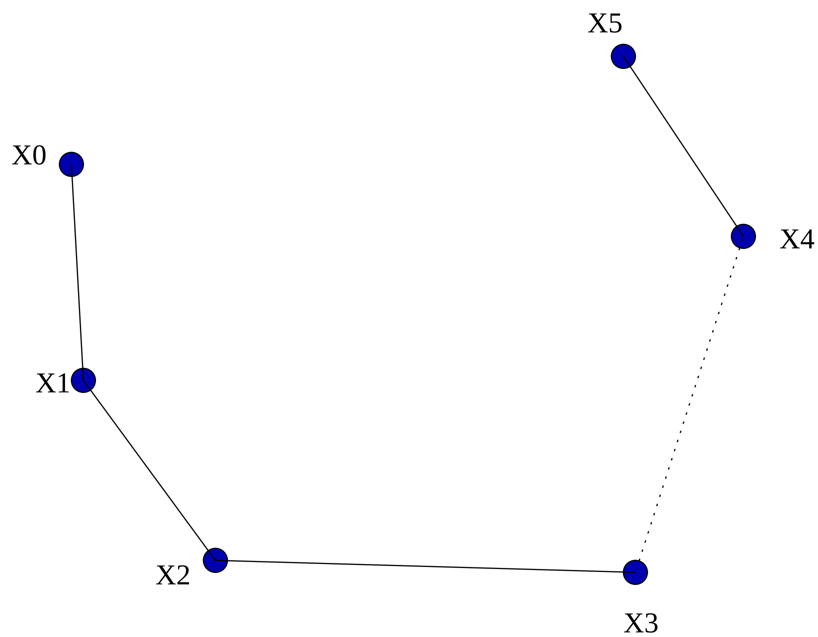


Figure 32: Three eye path steps and one light path step.

that $l = m + n + 1$, the extra step being the deterministic one connecting the eye path to the light path. What we need to calculate is the radiance along this path. We denote this by $\langle C_{m,n} \rangle$ meaning the contribution made by an eye path with m steps and a light path with n steps.We introduce two functions, L_e and L_l, for computing the product of transport operators along eye paths and light paths respectively.We define L_e as:

$$L_e(x_0, \Theta_{x0}, m) = \prod_{i=1}^{m} \left(\frac{f_r\,(x_i, \Theta_{xi}, \Theta_{xi-1})\, cos\Theta_{xi}}{pdf(\Theta_{xi})} \right)$$

and we define L_l as:

$$L_l(x_l, \Theta_{xl}, n) = \prod_{i=m+1}^{m+n} \left(\frac{f_r\,(x_i, \Theta_{xi}, \Theta_{xi-1})\, cos\Theta_{xi}}{pdf(\Theta_{xi})} \right)$$

where m, n, and l are as defined above..

This allows us to construct and expression for $\langle C_{m,n} \rangle$:

$$\langle C_{m,n} \rangle = \frac{L_e(x_0, \Theta_{x0}, m).g(x_m, x_{m+1}).L_l(x_l, \Theta_{xl}, n)}{d^2}.L_s \tag{67}$$

where d is the distance between x_m and x_{m+1} and g is the visibility function as before. This is very similar to the evaluation required for path tracing except for the following. Firstly we have a distance term to evaluate and secondly the PDFs at the points which connect the deterministic step will both be 1 since there is no probabilistic choosing of directions. We should also note that there will now be a PDF associated with the last point on the path. This point is on the light source and the PDF will be derived from whatever distribution was used to choose a direction from the light[11].

The equations above assume that we are using the probabilistically chosen point on the light path when carrying out direct lighting. As was already pointed out this is equivalent to using the $p1$ estimator and hence results in noisy images. It is far preferable to use the $p2$ estimator, i.e. choose points within the solid angle. We can't use this to choose an initial point on the light source however as this would necessitate knowledge of the position of the first point on the light path. We can however solve this problem as follows. When we have to evaluate a direct lighting path of the form $C_{i,0}$ we ignore the point chosen on the light source and simply temporarily choose a new one based on the solid angle subtended by the source from the point x_i.

We now move on to discuss the actual implementation. Implementing bidirectional path tracing requires a considerable amount more alteration to the basic ray tracing algorithm than does implementation of path tracing. The reason for this is that ray tracing is usually implemented in a recursive manner. This makes sense because we only need local BRDF information for each point on the path once and this can be easily retrieved on the way back up the recursion. After we have done the calculations for a particular point we can then discard all of its geometry and surface details and simply pass the relevant contribution back up. This, incidently, is the reason for the low memory requirements of ray tracing. Path tracing (in either direction) works in the same manner. However, with bidirectional path tracing a single point on a path might be part of several sub-paths and hence its geometry and surface details can be crucial to several radiance calculations. The recursive formulation won't work so easily in this case.

It is most intuitive to implement bidirectional path tracing as a two-stage process. The first stage consists of constructing the path and the second stage consists of selecting, weighting, and evaluating sub-paths of this path. As explained in the last section our basic implementation of BPT evaluates all possible sub-paths and weights all paths of equal length equally. The basic algorithm is as follows:

```
ShadePixel
    Colour set to zero
    Weights(1 ... (m+n+1)) set to zero
    Colours(1 ... (m+n+1)) set to zero
    path=GeneratePath(m,n)
    for(i=1;i<=n;i++)
```

[11] In our system we simply use a cosine distribution. More sophisticated directional light sources can be simulated by using more appropriate distributions.

```
        for(l=0;l<=m;i++)
            col=EvaluatePath(i,l,path)
            weights[i+l-1]++;
            colours[i+l-1] += col;
        endfor
    endfor
    for(i=0;i<(m+n);i++)
        if (weights[i]==0) weights[i]=1
        Colour += colours(i)/weights[i];
    endfor
```

The GeneratePath function traces a path of m steps from the eye and n steps from the light source and stored the result in a linked list. The function EvaluatePath takes this list as an argument, along with two integers i and j, and evaluates a sub-path of this path which consists of i eye steps and j light steps. It returns a colour value. If however the deterministic step is occluded, then EvaluatePath returns 0 for the colour. We increment the weight counter for paths of that length and add the colour.

We can now present some images which have been computed using BPT and compare them to those produced using PTNEE. Figure shows the results of running BPT on the same scene as the one used for figure 24. This scene poses problems for path tracing because it is lit primarily indirectly. Since the leftmost scene uses no light steps it is exactly equivalent to path tracing

(a) 8 eye steps, 0 light steps

(b) 4 eye steps, 4 light steps

(c) 1 eye step, 7 light steps

Figure 33: Results of running BPT on indirectly lit scene. All images computed at 1024 samples per pixel.

with next event estimation (and indeed can be verified as such by referring to figure 24). As we would expect on such a scene the more light steps which we take the better. The rightmost image was computed using 7 light steps and only 1 eye step and clearly manages to cope with the indirect illumination on the floor much better. These images also illustrate a common problem with bidirectional path tracing which is the presence of noise at points on surfaces which lie in close proximity to other surfaces. This is because deterministic steps which connect two closely adjoining surfaces necessitate the division by d^2 which in these cases will be very small. This division causes large radiance values for these points to be returned and hence we get bright spots. An easier defect to correct is the noise introduced by specular surfaces as illustrated in the sequence of images in figure 34. In this case we can see that larger number of light steps

(a) 8 eye steps, 0 light steps

(b) 4 eye steps, 4 light steps

(c) 1 eye step, 7 light steps

Figure 34: 64 samples per pixel. Scene is directly lit but most of it is not within the viewing frustum.

introduce significant noise as a result of the specular surfaces. This is to be expected since we are choosing reflection directions along the light path based on the specular reflection angle with respect to the previous point on the path. In other words we are following specular reflections *from* the light source. This intuitively seems like the right thing to do but it is not because these specular reflections will only contribute to the image significantly if they reach the eye. In other words whereas in the path tracing cases we are constructing specular-specular transport chains all the way to the eye, in the case above we are constructing specular-specular transport chains all the way to the last point on the eye path. The result is large blobs of specular illumination being transported around the scene and more than likely *not* being specularly reflected towards the eye. The images produced are extremely noisy. Lafortune correctly points out that a sensible strategy is to weight the paths so that we tend to favour specular transport when computing eye paths and diffuse transport when computing light paths (see next section)[12].

Finally let us construct a case where the scene is lit primarily by indirect lighting and where most of the scene is not visible from the current viewpoint. This is the type of situation which bidirectional path tracing should handle better than path tracing or light tracing. Figure 35 shows the scene in question from a viewpoint at the back wall.

We remove the light on the ceiling and place light sources within the green cones on the walls. This means that the scene is indirectly lit. We also change the viewpoint to that of figure 34 so that we are looking at the green ball on the table and hence most of the scene is not visible from the current viewpoint. The images shown in figure 36 are the results of running BPT on this scene using a variety of different combinations of eye and path lengths and a variety of different number of samples per pixel. As we can see BPT provides a definite improvement in this case as it has managed to construct smooth indirect illumination on surfaces such as the floor and the green sphere. However we still have a significant amount of noise, particularly in areas on, and

[12]An exception to this is if we are trying to compute caustics in which case we are explicitly looking for specular-diffuse light transport paths. It turns out that bidirectional path tracing is not a particularly efficient means of computing caustics and much better results can be obtained from the methods described in [23] and [24].

Figure 35: Image computed at 16 samples per pixel with path tracing.

adjacent to, specular surfaces. In the next section we discuss how to counter this by providing better ways of weighting the sub-path.

6.4 Weighting Sub-Paths Effectively

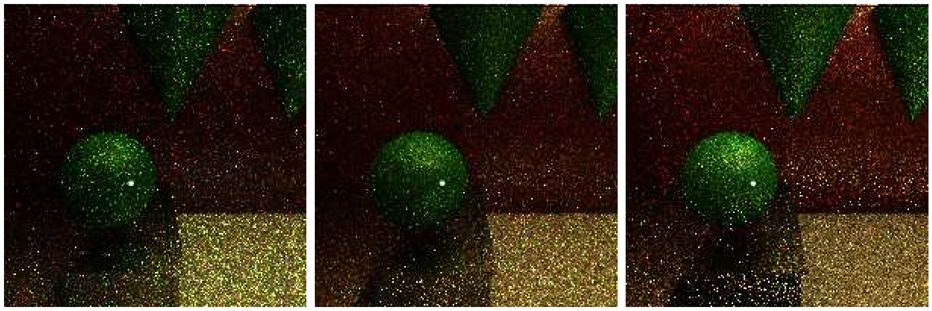
Uniform weighting of sub-paths causes significant noise when we have specular surfaces present as explained in the last section. Lafortune suggests an approach whereby weights are constructed for a given sub-path based on the specularity of the nodes and based on whether these nodes are part of an eye path or a light path. Simply put we want to favour specular eye paths and diffuse light paths. We denote the weight for a sub-path $C_{m,n}$ as $w_{m,n}$ (recall that $C_{m,n}$ denotes a path with m eye steps and n light steps). Note that we are free to assign any weights that we like to sub-paths as long as the following condition holds:

$$\sum_{i=0}^{N} w_{i,N-i} = 1 \tag{68}$$

for $N = 0, 1, ...Max$, where Max is the maximum path length. This means that for a given path length N, the sum of the weights for all sub-paths of this length must be one. We can easily use this idea to derive the basic path tracing algorithm if we set the weights as follows:

$$\begin{aligned} w_{i,j} &= \quad for\, j = 0,\, and \\ &= \quad 0\, otherwise. \end{aligned} \tag{69}$$

In practice this means that we evaluate all the sub-paths but only count the ones where there are no light steps. Firstly we define individual weights W_k for all points on the path. These weights should be selected based on the specularity of the surface. For highly specular surfaces the weights will be close to 1, for diffuse surfaces they will be close to 0. Lafortune then suggests



(a) m=8,n=0,s=64 (b) m=4,n=4,s=64 (c) m=1,n=7,s=64

(d) m=8,n=0,s=256 (e) m=4,n=4,s=256 (f) m=1,n=7,s=256

(g) m=8,n=0,s=1024 (h) m=4,n=4,s=1024 (i) m=1,n=7,s=1024

Figure 36: m is the number of eye steps,n the number of light steps, and s the number of samples per pixel.

selecting weights for sub-paths as follows:

$$
\begin{aligned}
w_{i,j} &= \prod_{k=0}^{i-2} W_k && for\, j = 0, \\
&= 0 && for\, i = 0,\, and \\
&= \left(\prod_{k=0}^{i-2} W_k\right)(1 - W_i) && otherwise
\end{aligned}
\tag{70}
$$

These give significantly better results, particularly for scenes with highly specular surfaces.

Veach and Guibas [22] present a more theoretically sound means of weighting estimators as part of a general framework for combining Monte Carlo estimators. Suppose we are integrating some function and we have a number of pdf's at our disposal which we can use to carry out importance sampling. We shall call these estimators and denote them as $p_1, p_2, ..., p_n$. We can adopt a number of different strategies in this case. The first strategy is to simply choose one of the pdf's, use this and ignore the other ones. In many cases this is the best approach, however, there may be cases where one estimator is a close fit to part of the integrand and another is a close fit for the other part of the integrand. In these cases we would like to take advantage of both. So, the second possibility is to use all of the estimators and divide the results by n. This should *not* be confused with averaging primary estimators to obtain a secondary one. In the former case we have n different pdf's and to obtain a primary estimator we sample according to all of them and then divide by n to get the average. This becomes our primary estimator. In the latter case we take N of the these primary estimators and average them in order to get a secondary estimator. This can work reasonably well but is not optimal.

The third possibility, as proposed by Veach and Guibas, is the so-called *balance heuristic*. This aims for an optimal combination of estimators.We select weights according to the following formula:

$$
w_i(x) = \frac{p_i(x)}{\sum_{j=1}^{n} p_j(x)} \tag{71}
$$

where $p_i(x)$ is the probability of choosing x using pdf p_i. So the weight for a particular sample is the probability of generating that sample according to the probability distribution which was actually used to generate it, divided by the sum of the probabilities of generating that samples according to all the alternative probability distributions with which *it could have been* generated. This means that if a sample is chosen which is very likely with respect to it's corresponding pdf but unlikely with respect to others it will be assigned a weight close to 1. If a sample is chosen which is unlikely with respect to its pdf but very likely with respect to some other pdf then it will get a low weight. If a sample is chosen which has a reasonably similar chance of being chosen by all distributions then the effect will be similar to simply averaging it. Veach points out that this technique will not turn several poor estimators into one good one but rather will guarantee that, given some set of estimators we are using, it will provide the optimal means of combining them.

We can apply this to bidirectional path tracing as follows. As we explained previously, for a given path length, we are presented with a number of different potential primary estimators. For example, we can construct a path of length 4 by taking two steps from the light source and one from the eye, by taking 1 from the light and 2 from the eye, or by taking 3 from the eye. Each of these constitutes an estimator in the sense described above. Again we can combine them in three different ways. Firstly we can just use one of them. In the bidirectional case this would be equivalent to simply joining the eye path with the light path and not connecting up anything else. Secondly we could average them. This would mean that we connect up all sub-paths, evaluate the flux along all of them, and average the results. This is the technique that our present implementation employs, i.e. uniform weighting. The third possibility, and the one that Veach suggests, is to just use one of the available estimators but *weight it using the balance heuristic.* This could work as follows. Start by taking a step from the eye, as shown in diagram 37, and choosing a point y on the light source. Note that in the following discussion we are using Veach's

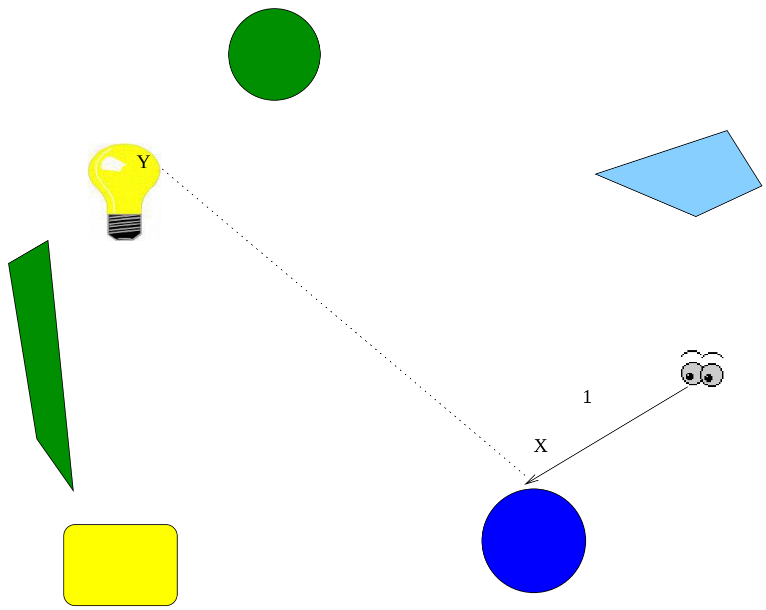


Figure 37: 1 eye step

convention of regarding the choice of a point on the light source as constituting a step. Let us refer to the estimator which takes m steps from the eye and n steps from the light source as $p_{m,n}$ and the intersection point as x. We connect x with the y on the light in order to get a path of length 2. Our sample is now the path $e-x-y$ and was generated using estimator $p_{1,1}$. This path however could have been generated using a different estimator, namely $p_{2,0}$, or alternatively it could be generated with $p_{0,2}$. So, we compute the flux radiance along this path (which is independent of the estimator used to generate it) and then compute a weight as follows:

$$w_2(e-x-y) = \frac{p_{1,1}(e-x-y)}{p_{1,1}(e-x-y)+p_{2,0}(e-x-y)+p_{0,2}(e-x-y)} \tag{72}$$

So, the weighting is based on, not just the probability of generating the path using the current estimator, but also the probability of generating it using all the other estimators at our disposal. We then move on by extending another ray from the end of the light path or from the end of the eye path (or both) and continue in this fashion. A convenient side-effect of this, from an algorithmic point of view, is that it leads us to a recursive formulation of bidirectional path tracing.

7 Conclusions and Future Work

7.1 Summary

In this report we have discussed a general class of Monte Carlo rendering techniques which use random walks to estimate the values of recursive integrals such as the rendering equations. These types of random walks are typically referred to as Markov chains in the statistics literature. We started by providing a overview of the subject of Monte Carlo integration before moving on to describe the general global illumination problem, and the rendering equation in particular. We then presented three different means of solving the rendering equation using random walk type Monte Carlo algorithms. These three methods were pure path tracing, path tracing with next event estimation, and bidirectional path tracing. In each case we provided the mathematical and algorithmic details, discussed our implementation, and presented some results.

In this final section we describe a different means of applying Markov chains to the global illumination problem. The usual way of using Markov chains can be summarised as follows. Each consecutive point on the chain represents a sample of the next level of recursion of the rendering equation (or the potential equation if we are going from the light source). The end-result if the construction of a path through the integral space where each point on the path is probabilistically determined from the previous one. So we use Markov chains to construct paths by choosing a new point on the path based on the previous one. An alternative strategy is to regard a complete path as a sample in itself and use Markov chains to generate entirely new *paths* based on the previous ones in the chain. We can then estimate the flux along these new paths and scale them according to the probability of their generation. The first application of this to rendering was by Veach [24] where he used the well-known Metropolis sampling algorithm in this manner. In the next section we describe this work. Metropolis is part of a family of sampling algorithms which may be employed for this purpose. In the final section we briefly introduce some of these sampling algorithms and finish by describing our plans to incorporate these algorithms into a rendering context.

7.2 The Metropolis Algorithm

As explained previously bidirectional path tracing is designed for those situations where difficult lighting conditions prevail. Either the lighting is primarily indirect, or the bulk of the scene is not visible from the viewpoint, or both. It deals with these situations by extending paths from the light and from the eye and connecting them. However, after a path has been generated and the appropriate fluxes and probabilities computed, it is discarded, whereupon a new path is commenced. Intuitively this seems wasteful. The algorithm goes to a lot of trouble to establish a complete path from eye to light source and then discards it. It seems likely that, in close vicinity to this path, we will find many other complete light transport paths also. However the bidirectional algorithm is no more likely to try to use these than it will any other possible paths. It doesn't use the information gained from the previous step to guide the next one.

Veach explains how to use the Metropolis sampling algorithm to generate a path based on the existing one. In order to do this we firstly re-formulate the global illumination problem as an integration problem over a space of paths and therefore a light transport path is a sample from this space. New samples are obtained by *mutating* the current one. The flux along the new path is computed and scaled according to the probability of the path's generation (this is called the *transition probability*). Each mutation can be rejected however according to a carefully chosen *acceptance probability*. The correct choice of acceptance probability ensures that the samples converge to the correct distribution. As each path is sampled we update the image on the screen. Veach presents the basic algorithm as follows:

```
Metropolis
x = InitialPath()
image = array of zeros
for i = 1 to N
    y = Mutate(x)
    a = AcceptProb(y|x)
    if RANDOM() < a
        then x = y
        RecordSample(image,x)
return image
```

The initial path can be generated in any way but it is suggested that bidirectional path tracing is the ideal means of doing this. Veach's paper goes into detail as to how to choose the transition and acceptance probabilities. The results of this are extremely impressive and Veach reports considerable improvement in rendering times on scenes with difficult lighting conditions (for example a room which is lit entirely by a slightly ajar door into another room which contains a light source). It is still open to question however how well the algorithm performs in general and in particular on scenes which do not exhibit the required characteristics. Some recent work [20] has presented results which seem to confirm this.

7.3 Future Work

It seems that the only feasible way of reducing the computational demands of Monte Carlo methods is to design better importance sampling strategies which more effectively sample the integrands in question. Most existing techniques only do *local* importance sampling. For example, they use the BRDF of the current surface as a PDF. This means that they are only taking into account a limited amount of information. It is likely that better strategies can be developed which carry out *global* importance sampling. These techniques should sample based on PDF's which are derived from information regarding the entire path, or indeed an entire space of paths. Also it seems obvious that these techniques should be *adaptive*, meaning that they should hone and modify their weights as they sample an environment in order to progressively improve the accuracy of the sampling. The best example of these ideas which has been published to date is Veach's Metropolis algorithm as presented in the previous section. However it seems likely that there are many more techniques which may prove similarly effective and may indeed overcome some of the limitations of Metropolis as pointed out in [20]. So, future work is focused on two themes:

- Implementation and evaluation of the Metropolis algorithm.

- Identification of alternative sampling strategies (e.g. Multiple-Try Metropolis, Gibbs Sampling, variations of Simulated Annealing), evaluation of same in terms of their suitability to global illumination, followed by implementation and evaluation.

References

[1] S.Collins. *Adaptive Splatting for Specular to Diffuse Light Transport.* In *Fifth Eurographics Workshop on Rendering*: ps 119-135. Darmstadt, Germany, June 1994.

[2] R.L. Cook, T. Porter, L.Carpenter. Distributed Ray Tracing. *Computer Graphics,* 18(4): 165-174, July 1984. ACM SIGGRAPH '84 Conference Proceedings.

[3] P.Dutre, E.Lafortune, Y.D.Willems. *Monte Carlo Light Tracing with Direct Computation of Pixel Intensities.* In *Compugraphics '93,* ps: 128-137, Alvor, 1993.

[4] P. Dutre. *Mathematical frameworks and Monte Carlo algorithms for global illumination in computer graphics.* Ph.D. thesis. 1996

[5] C.Goral, K.E.Torrance, D.P.Greenberg, B.Battaile. Modeling the interaction of light between diffuse surfaces. *Computer Graphics,* 18(3):213-222. July 1984.

[6] H.Gouraud, *Computer Display of Curved Surfaces.* Technical Report. Department of Computer Science, University of Utah. 1971

[7] D.Hearn, M.P.Baker. *Computer Graphics.* 2nd edition. Prentice Hall. 1997

[8] D.S.Immel, M.F.Cohen, D.P.Greenberg. A Radiosity Method for Non-diffuse Environments. *Computer Graphics,* 20(4):133-142, August 1986. ACM SIGGRAPH '86 Conference Proceedings.

[9] H.W.Jensen. *Importance Driven Path Tracing using the Photon Map.* In *Rendering Techniques '95 (Proceedings of the Sixth Eurographics Workshop on Rendering).* ps:326-335. Dublin, 1995.

[10] J.T.Kajiya. The Rendering Equation. *Computer Graphics*, 20(4):143-150, August 1986. ACM SIGGRAPH '86 Conference Proceedings.

[11] M.Kalos, P.Whitlock, *Monte Carlo Methods: Volume 1: Basics.* Wiley & Sons. 1986

[12] E.P.Lafortune, Y.Willems. *Bi-directional Path Tracing.* In *Compugraphics '93,* ps: 145-153, Alvor, 1993.

[13] E.P. Lafortune, Y.D. Willems. *Using the Modified Phong Reflectance Model for Physically Based Rendering.* Report CW 197. Department of Computing Science, K.U. Leuven. November 1994.

[14] E.P.Lafortune. *Mathematical Models and Monte Carlo Algorithms for Physically Based Rendering.* Ph.D. thesis. 1996

[15] S.Pattanaik. *Computational Methods for Global Illumination and Visualisation of Complex 3D Environments.* PhD thesis, NCST Birla Institute of Technology and Science, Pilani, India, Feb 1993.

[16] B.T.Phong, Illumination for Computer-Generated Images. *Communications of the ACM* 18(6): 311-317. 1975

[17] P.Shirley, *Physically Based Lighting Calculations for Computer Graphics.* PhD thesis, University of Illinois, Nov. 1990.

[18] P.Shirley, C. Wang. Direct lighting calculation by monte carlo integration. In *Eurographics Workshop on Rendering.* 1991

[19] L.Szirmay-Kalos. *Stochastic Methods in Global Illumination: State of the Art Report.* Technical Report/TR-186-98-23. Department of Control Engineering and Information Technology, Technical University of Budapest. August '98.

[20] L.Szirmay-Kalos, P.Dornbach, W.Purgathofer. *On the Start-Up Bias Problem of Metropolis Sampling.* Proceedings of the 8th International Conference in Central Europe on Computer Graphics, Visualization, and Interactive Digital Media '99. Plzen, Czech Republic.

[21] E.Veach, L.Guibas. *Bidirectional Estimators for Light Transport.* In *Computer Graphics (SIGGRAPH '95 Proceedings),* ps: 419-428, 1995.

[22] E.Veach, L.Guibas. *Optimally combining sampling techniques for light transport.* In *Computer Graphics,* vol. 29, pp 419-428, August 1995.

[23] E. Veach. Non-Symmetric scattering in light transport algorithms. In *Eurographics Workshop on Rendering.* 1996.

[24] E.Veach. *Robust Monte Carlo Methods for Light Transport Simulation.* Ph.D. thesis. 1997

[25] T.Whitted. An improved illumination model for shaded display. *Communications of the ACM* 23(6):343-349. 1980