Chair of Computer Graphics - Albert-Ludwigs-University of Freiburg

---

Master's Thesis
Summer Semester 2023

Implementation and Analysis of Path Tracing using Multiple Importance Sampling

Submitted by Adil Rabbani
Supervised by Prof. Dr.-Ing. Matthias Teschner
Second examiner: Prof. Thomas Brox

Declaration

I hereby declare, that I am the sole author and composer of my thesis and that no other sources or learning aids, other than those listed, have been used. Furthermore, I declare that I have acknowledged the work of others by providing detailed references of said work.
I hereby also declare, that my Thesis has not been prepared for another examination or assignment, either wholly or excerpts thereof.

Freiburg, 04/29/2023

Place, Date

Signature

Contents

1 Acknowledgements

This thesis would not have been possible if I was working on it alone. I thank my supervisor **Prof. Dr.-Ing. Matthias Teschner** for the discussions and constant support throughout the duration of this thesis. I would like to thank my parents and my family for their support and for encouraging me to pursue my dreams even if that meant living thousands of kilometers away from them.

2 Abstract

Creating synthetic computer generated images that mimic physical interaction of how light travels in a virtual scene, has a great deal of importance not only in the field of computer graphics but in computer science in general. These physically accurate computer generated images have applications in movies, CAD softwares and augmented as well as virtual reality simulations. The current advancement in computer hardware has enabled us to move away from rasterization based solutions to physically based Monte Carlo methods to create such highly realistic images. This also opens up new areas of research in light transport simulation in order to create methods that can handle a variety of virtual scenes robustly.

In this thesis, we discuss one of the widely used monte carlo method for light transport simulation namely *path tracing*, starting from the very basic concepts of light modeling as building blocks and arriving at a monte carlo estimator that approximates the rendering equation. We observe that such a method is able to reproduce effects such as global illumination and soft shadows while also being able to model different types of materials we observe in real life. We further analyze the estimator by testing it in different scenarios in order to gain more insights.

3 Introduction

One primary goal in the field of computer graphics is to be able to create virtual environments that are indistinguishable from real life environments. A vast amount of research is being done to increase the realism in computer generated images. On the other hand, it is also necessary to achieve these computer generated images (or renders) in a reasonable amount of time so a huge amount of research is also being done to speed up the process of creating such images. This brings us to an algorithm called **ray tracing** which is one of the methods to create such physically accurate images. Ray tracing employs principles from **geometric optics** which is a model of optics that describes light propagation in terms of rays [44].

We are given a description of the scene, including the geometry and the scattering properties of surfaces. We are also given a description of the light sources and the viewpoints from which our image should be generated. We start with casting rays from each of the pixels of our screen from the view position. And as we do this, we hit various objects in our scene. Objects in a virtual scene are made of geometrical shapes. It is a common practice in computer graphics to use triangles as the base primitive to make up complex models. For instance, a fairly reasonable looking bunny [35] can be created using about 69.5k triangles. Now since every object in our scene can be created by using triangles, we need a way to intersect our rays with these triangles in our scene. This is done by ray-primitive intersection tests. As an object is hit, we compute illumination at that point depending on how far the position is from a light source. As we also want to create shadows in our scene, when a point in our scene is hit by a ray, we shoot another ray from that position but this time, towards the light source. If this ray hits another object on its way towards the light source, we conclude that the point we are shading is in a shadow. If one object is in front of another object, we shade the object which is closest from the ray origin.

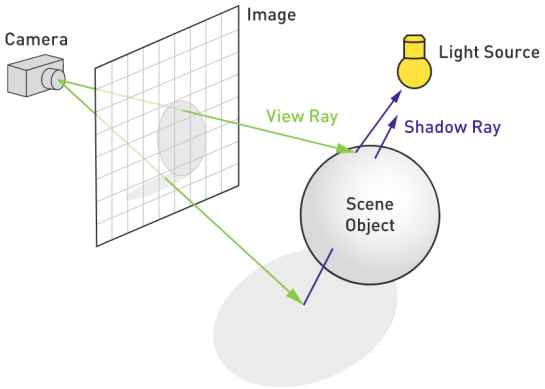


Figure 1: Visualization of how ray tracing works. [23]

This process of sending rays through each pixel, evaluating ray-primitive intersection tests, and computing illumination at each point in our scene is called **ray tracing** which was originally introduced for the first time by Turner Whitted [42]. The only problem with this original method was that, while it was able to demonstrate reflections, refractions and direct illumination from light sources, it was still not able to reproduce some commonly seen effects in photographs captured by cameras such as motion blur, soft shadows and indirect illumination (illumination which is bounced off from different surfaces in our scene). The next breakthrough in photorealistic rendering came with the introduction of **stochastic ray tracing** [14] which helped create the previously mentioned effects in a more practical manner. Just two years after that, James Kajiya formulated light traveling in a virtual scene into an integration problem also known as the **rendering equation**, [27] which enabled us to create highly realistic images that also support indirect illumination. An algorithm that approximates the rendering equation is commonly known as a **light transport algorithm** or **LTE** in short.

A *light transport algorithm* is what decides how the illumination in our virtual scene is carried out. It answers questions like how the light rays sent towards different types of objects in our scene would illuminate those objects and bounce off them or get absorbed by them. So as the name suggests, a light transport algorithm is what helps us generate realistic images by simulating the emission and scattering of light in an artificial environment [39]. This requires us to solve the so-called *rendering equation* which will be discussed later in more detail. But what is important to mention here is that the rendering equation helps us establish connections between light sources, camera and surfaces in our scene and compute the amount of light (or radiance) reaching at those points along these connections as accurately as possible.

Path tracing is one of the light transport algorithms which is extensively being used in computer graphics. It has applications in computer generated movies as well as interactive applications such as video games. Some famous path tracing softwares include Pixar's Renderman [12] and Disney's Hyperion [9]. What makes path tracing suitable for such applications is the fact that it generates images which are **unbiased**. A method is *unbiased* if it produces the correct answer on average. Why is this important? Because if we have a method which is biased, it might produce images which might favor one scene over another. We might start experiencing artifacts for some scenes while other scenes might look flawless. Some scenes might seem darker in some lighting conditions while being brighter in other conditions. Therefore, it is important for light transport algorithms to be robust in different lighting conditions and also produce their results within acceptable time-bounds. Both of these problems i.e robust performance and acceptable time-complexity is handled nicely by path tracing since it uses *Monte Carlo Integration* (also discussed later in more detail), to solve the rendering equation.

However, a well known problem when using MC Integration is that it approximates the integral we are trying to solve which only converges to an acceptable solution given a huge number of samples. A vast amount of research is being done to use monte carlo integration efficiently so that we are able to use as few samples as possible while also reducing variance at the same time. One of the techniques that is being used to tackle this problem is **multiple importance sampling (MIS)** [39].

It is also important to analyze monte carlo estimators, so that we're able to see how robust they are and discuss ways to improve them even further. For this purpose, different metrics help us to see if our image converges to the ground truth. All of these aspects will be discussed in this thesis, from the concepts of modeling light and materials in our virtual scenes to approximating rendering equation using monte carlo integration and then analyzing it further.

We will now go through some concepts which are important to discuss before we move on towards our implementation.

4 Concepts

4.1 Radiometric quantities

Since we need to measure light traveling in a scene while also obeying the laws of conservation, it is important to first describe some physical quantities that helps us quantify the amount of light arriving at a particular point in scene.

4.1.1 Radiant flux

We will start with describing **Radiant flux**, often denoted by Φ. It is defined as the total light energy passing through a surface per unit time.

$$\Phi = \frac{dQ}{dt} \tag{1}$$

It is measured in Watts (W) or Joules/seconds (J/s). This however is not descriptive enough for our purpose. The reason is the fact that we have no idea of knowing if a high amount of flux was recorded over a huge area or a low amount of flux arrived at a small area because this would result in the same amount of radiant flux.

4.1.2 Irradiance

This motivates us to introduce another radiometric quantity known as **Irradiance**, denoted by E. Irradiance is the amount of radiant flux per unit area.

$$E(x) = \frac{\Phi}{A} \tag{2}$$

It is measured in Watts/meter squared (W/m^2). However, this again, is still less descriptive for our purpose since we still don't know at which angle is the incident light coming from. So, this can mean that we can have a high amount of energy coming from a wide angle or the same energy coming from a small angle, resulting in the same amount of irradiance.

4.1.3 Radiance

This finally brings us to a radiometric quantity known as **Radiance**, denoted as L. It is the radiant flux per unit area and solid angle given by:

$$L(\boldsymbol{x} \leftarrow \omega) = \frac{d^2\Phi}{dAcos\theta d\omega} \tag{3}$$

It is measured in Watts/meter squared.steradians (W/m^2sr). A **steradian** is a unit for solid angle. In the above equation, $d\omega$ is the differential solid angle given by:

$$d\omega = \frac{dA(\boldsymbol{x})cos\theta_{\boldsymbol{x}}}{r_{\boldsymbol{x}}^2} \tag{4}$$

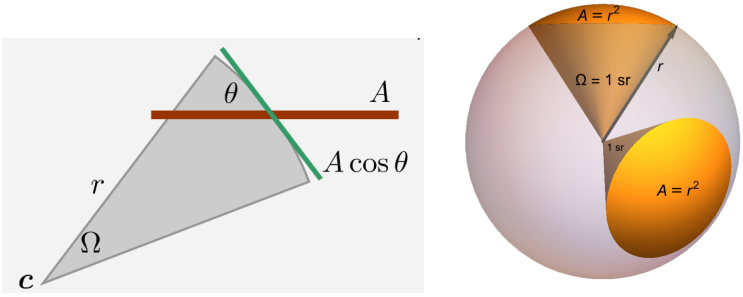


Figure 2: Left) Solid angle of an arbitrary surface. [38]. Right) Any area on a sphere which is equal in area to the square of its radius, when observed from its center, subtends precisely one steradian [3].

Where, $dA(\boldsymbol{x})$ is an infinitesimally small area at position $\boldsymbol{x}$. A solid angle is a measure of the amount of field of view from some particular point that a given object covers. A nice intuition for solid angle is that it can be considered a 3D variant of an arc in 2D. So, in 2D an arc defines a part of a circle; and in the same way in 3D, a solid angle defines a patch on a sphere. A solid angle of an arbitrary surface is given by:

$$\Omega = \frac{Acos\theta}{r^2} \tag{5}$$

4.2 Emission, Propagation and Scattering of light

As we already stated before, ray tracing involves employing principles from geometric optics. We can think of light as small light particles and every light particle starts from a light source and travels in a straight line. It will eventually hit a surface and upon interaction this particle would either be absorbed or scattered in another direction and continue bouncing around the scene. We now briefly describe how emission, propagation and scattering of these particles or **photons** is formulated.

4.2.1 Emission

Light emitting from a point $\boldsymbol{x}$ in the direction $\boldsymbol{\omega}$ is mathematically represented as $L_e(\boldsymbol{x} \rightarrow \boldsymbol{\omega})$. This can be a point light in our scene emitting photos with a specific amount of flux or it can also be an area light such as a light bulb which emits these photos from a specific amount of area in our scene.

4.2.2 Propagation

It is important that law of energy conservation is obeyed with the light transport in our scene. The amount of flux leaving a surface point $\boldsymbol{x}$ should be the same amount of flux arriving at a particular surface point $\boldsymbol{y}$. Mathematically,

$$L(\boldsymbol{x} \rightarrow \boldsymbol{\omega_{xy}}) = L(\boldsymbol{y} \leftarrow \boldsymbol{\omega_{yx}}) \tag{6}$$

4.2.3 Visibility

It is important to have a function that can tell if two surfaces are mutually visible to each other. In complex scenes, it is possible that various objects are close to each other or in fact obstruct the light traveling because one object is close to the light source and obstructs the light reaching the other object. So before even doing any computation for radiance reaching a point we can first check if the visibility function returns true for two specific points.

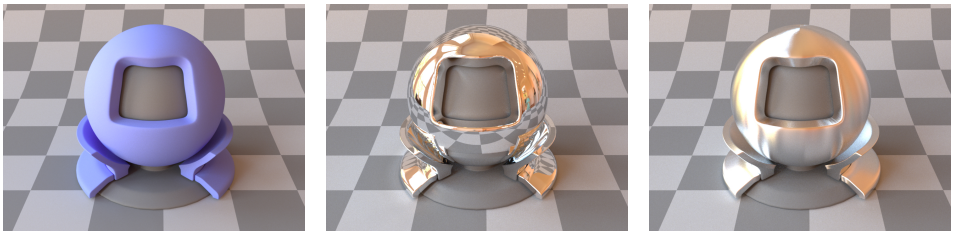
$$V(\boldsymbol{x}, \boldsymbol{y}) = \begin{cases} 1 & if\ \boldsymbol{x}\ and\ \boldsymbol{y}\ are\ mutually\ visible, \\ 0 & otherwise \end{cases} \tag{7}$$

4.2.4 Modeling materials

All objects in our scene are made up of some kind of material. Among the various types of materials used in computer graphics the most common ones are **diffuse**, **specular** and **glossy** materials.

4.2.4.1 Common materials

Diffuse materials are materials that resemble a rough/matte surface. A real life example of a diffuse material could be a painted cement wall or a desk made of wood but not polished enough to see any kind of reflections. The material closely resembles the material shown figure 3(a). Specular material on the other hand is a material that resembles highly reflective and shiny surfaces. This could be a highly polished metal in which you can see your own reflection just like seeing yourself in a mirror. The material closely resembles the material shown in figure 3(b). A glossy material is a mixture of both diffuse and specular materials. This resembles closely to a plastic like material which is reflective in some parts but also diffuse at some. Another example of this kind of material is a metal surface that is reflective but also roughed out. The material closely resembles the material shown in figure 3(c).



(a) Diffuse material (b) Specular material (c) Glossy material

Figure 3: Diffuse, specular and glossy materials from left to right. [24]

In order to model these materials, we need to model how the incoming light $\boldsymbol{\omega}_i$ is reflected from a point $\boldsymbol{p}$ on an object into the outgoing direction $\boldsymbol{\omega}_o$. This function that takes in incoming light direction and a point on a surface and returns the outgoing light direction is known as the **Bidirectional Reflectance Distribution Function** or **BRDF**.

4.2.4.2 Bidirectional Reflectance Distribution Function (BRDF)

Formally, BRDF is the ratio of outgoing radiance towards $\boldsymbol{\omega}_o$ and irradiance due to incoming radiance from $-\boldsymbol{\omega}_i$. So, BRDF returns the percentage of light leaving a point on a surface taking into account the incoming light from a particular direction. It is commonly denoted as f_r and represented as:

$$f_r(\boldsymbol{\omega}_i, \boldsymbol{p}, \boldsymbol{\omega}_o) = \frac{dL_o(\boldsymbol{p}, \boldsymbol{\omega}_o)}{dE_i(\boldsymbol{p}, \boldsymbol{\omega}_i)} \tag{8}$$

So in order to model the materials we discussed previously, we model how light is reflected from those materials. For a diffuse surface, light is reflected in all directions equally, for a specular material, light is reflected to exactly one direction. On the other hand, for a glossy surface, more light is reflected in some directions than other.

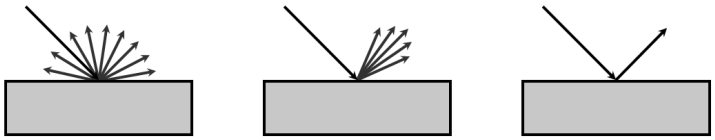


Figure 4: How light reflects from diffuse, glossy and specular materials. [33]

It would be nice to discuss a diffuse BRDF here since it is not that complicated. We will skip the derivation of a diffuse BRDF, but formally it is given by:

$$f_r(\boldsymbol{\omega}_i, \boldsymbol{p}, \boldsymbol{\omega}_o) = \frac{\boldsymbol{\rho}}{\pi} \tag{9}$$

where $\boldsymbol{\rho}$ is known as the **albedo** or the reflectance coefficient. This coefficient describes the amount of color reflected and the amount absorbed by a value between 0 and 1 for each red (R), green (G) and blue (B) channels. For instance, a value of $(1, 1, 0)$ describes a yellow surface, where red and green are reflected and blue is absorbed. Equation (9) also tells us that the BRDF for diffuse materials is independent from incident and exitant light $\boldsymbol{\omega}_i$ and $\boldsymbol{\omega}_o$. The BRDF also have some strict properties which are given as follows:

1. **Helmholtz reciprocity**: Direction of the ray of light can be reversed. This means that even if we switch the incident and exitant directions, the value for the BRDF remains the same. Mathematically:
$$\forall \boldsymbol{\omega}_i, \boldsymbol{\omega}_o : f_r(\boldsymbol{\omega}_i, \boldsymbol{p}, \boldsymbol{\omega}_o) = f_r(\boldsymbol{\omega}_o, \boldsymbol{p}, \boldsymbol{\omega}_i) \tag{10}$$

2. **Positivity**: It is not possible for a BRDF value to be negative. This is because incident flux is quantified as a postive number and so exitant flux should also be a positive number. Mathematically:
$$\forall \boldsymbol{\omega}_i, \boldsymbol{\omega}_o : f_r(\boldsymbol{\omega}_i, \boldsymbol{p}, \boldsymbol{\omega}_o) \geq 0 \tag{11}$$

3. **Energy Conservation**: An object that absorbs or reflect light cannot reflect more than the incident amount. This is an energy conservation property which makes sense because an object which receives light can at most reflect the same amount of light but cannot emit more than that. Mathematically:
$$\int_{\Omega} f_r(\boldsymbol{\omega}_i, \boldsymbol{p}, \boldsymbol{\omega}_o)\ cos\theta_o\ d\boldsymbol{\omega}_o \leq 1 \tag{12}$$

Since now we have a good understanding of how to model light and how to model some common materials in our virtual scene, we can move forward to discuss the most important equation that combines everything and solves the light transport problem i.e the **Rendering Equation**.

4.3 Rendering Equation

In a virtual scene, an object can emit light itself. One example of such emissive material is a light bulb. But an object can also receive light from different directions. This is analogous to the materials we have in our scene and how they received and reflect light to different parts of our scene. So as this light reaches an object, it would also reflect or absorb some light. This phenomenon is compactly represented by the **Rendering Equation** [27].

4.3.1 Hemispherical Form

The rendering equation is formulated as follows:

$$L_o(\boldsymbol{p}, \boldsymbol{\omega}_o) = L_e(\boldsymbol{p}, \boldsymbol{\omega}_o) + \int_\Omega f_r(\boldsymbol{\omega}_i, \boldsymbol{p}, \boldsymbol{\omega}_o)\ L_i(\boldsymbol{p}, \boldsymbol{\omega}_i)\ cos\theta_i\ d\boldsymbol{\omega}_i \tag{13}$$

The above equation states that the amount of light leaving the surface or the outgoing radiance at a point $\boldsymbol{p}$ is equal to the emitted radiance L_e at that point plus the reflected incoming light attenuated by the angle between the incoming direction and the normal at that point. Note that here $d\boldsymbol{\omega}_i$ is the differential solid angle for the incoming direction. The solid angle subtended by the surface element $d\boldsymbol{x}$ is the fractional area of a sphere of radius r taken up by the projected area $d\boldsymbol{x}_p$ of $d\boldsymbol{x}$:

$$d\boldsymbol{\omega_i} = \frac{d\boldsymbol{x}_p}{r^2} cos\theta d\boldsymbol{x} \tag{14}$$

The integral on the right hand side without the light emitted is called the **reflectance integral**. One problem is the fact that the incoming radiance L_i at some point would in fact be exitant radiance L_o because it might have been reflected from another surface in the scene which makes this equation recursive.

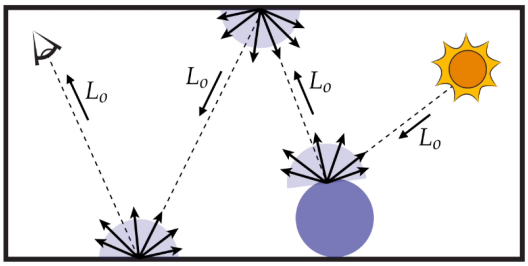


Figure 5: Recursive nature of the rendering equation. [17]

This means that light traveling in straight lines as rays, starting its journey from a light source would eventually hit another object in a scene which again would reflect light according to the materialistic properties it has and then that light will soon be received by another object. This process keeps on happening until this light eventually hits the camera sensor in our scene. For this reason, the rendering equation is **recursive**. This also makes the equation difficult to solve since different materials in our scene can have different BRDFs and can make the integral impossible to compute in closed form. We will look into how rendering equation can be approximated by a technique known as **monte carlo integration** but first we need to go through another form of the rendering equation which is well suited when we want to integrate over an area instead of integrating over a solid angle.

4.3.2 Area Form

The incoming radiance L_i in our rendering equation (13) is a recursive term since that radiance can infact be an outgoing radiance reflected from other surface in our scene. One practical form of writing

this term L_i is to write it as an outgoing radiance reflected from another point in the direction $-\boldsymbol{\omega}_i$. In order to do this, we trace a ray from the point $\boldsymbol{p}$ into the direction $\boldsymbol{\omega}_i$ and ask at the point $\boldsymbol{p'}$ intersected, how much radiance is going out in the direction $-\boldsymbol{\omega}_i$. So, our term L_i can be written as:

$$L_i(\boldsymbol{p}, \boldsymbol{\omega}_i) = L_o(t(\boldsymbol{p}, \boldsymbol{\omega}_i), -\boldsymbol{\omega}_i) = L_o(\boldsymbol{p'}, -\boldsymbol{\omega}_i) \tag{15}$$

And so for a sample leaving the view (camera) position we intersect a point on a surface in our scene and trace a ray from that point in another direction. We evntually hit another surface and keep doing the same until we hit an emissive surface. This emissive surface has the emissive value L_e as in the rendering equation (13), which is also the base condition for our sample. This emissive value is then treated as the outgoing radiance L_o leaving from that point. And as the equation is recursive, we eventually also have the incoming radiance L_i at all the surfaces hit indirectly. In this way, we have traced a sample from our view to a light source.

In the area form of the rendering equation, we are focused on treating the reflectance integral in the rendering equation as an integral over area instead of an integral over directions of the sphere (solid angle). This form also sheds more light into the recently introduced ray casting operator and provides a more compact way to represent the rendering equation.

So far, our rendering equation is in the form,

$$L_o(\boldsymbol{p}, \boldsymbol{\omega}_o) = L_e(\boldsymbol{p}, \boldsymbol{\omega}_o) + \int_\Omega f_r(\boldsymbol{\omega}_i, \boldsymbol{p}, \boldsymbol{\omega}_o)\ L_o(\boldsymbol{p'}, -\boldsymbol{\omega}_i)\ cos\theta_i\ d\boldsymbol{\omega}_i \tag{16}$$

Consider the figure shown below, where $\boldsymbol{p'}$ is the position where we compute the outgoing radiance towards a position $\boldsymbol{p}$. The incoming radiance is from another position $\boldsymbol{p''}$:

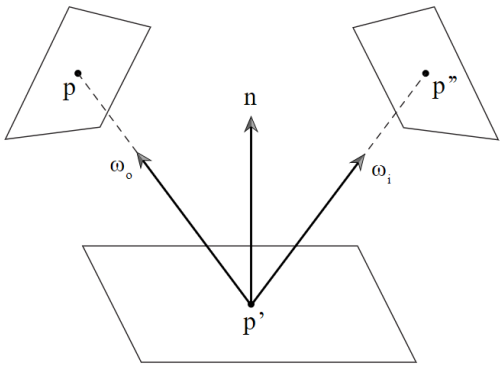


Figure 6: Three point form (area form) of the rendering equation. [32]

We can now define the emissive term L_e, the BRDF f_r and the incoming radiance as:

$$L_e(\boldsymbol{p}, \boldsymbol{\omega}_o) = L_e(\boldsymbol{p'} \to \boldsymbol{p}) \tag{17}$$

$$f_r(\boldsymbol{\omega}_i, \boldsymbol{p}, \boldsymbol{\omega}_o) = f_r(\boldsymbol{p''} \to \boldsymbol{p'} \to \boldsymbol{p}) \tag{18}$$

$$L_o(\boldsymbol{p'}, -\boldsymbol{\omega}_i) = L(\boldsymbol{p''} \to \boldsymbol{p'}) \tag{19}$$

We also need to introduce a geometric coupling term G in our equation. The geometric term is also coupled with the visibility function (7):

$$G(\boldsymbol{p''} \leftrightarrow \boldsymbol{p'}) = V(\boldsymbol{p''} \leftrightarrow \boldsymbol{p'}) \frac{|cos\theta'||cos\theta''|}{||\boldsymbol{p''} - \boldsymbol{p'}||^2} \tag{20}$$

where θ'' is the angle between $\boldsymbol{\omega}_i$ and normal of $\boldsymbol{p''}$ ($\boldsymbol{n}_{\boldsymbol{p''}}$) and θ' is the angle between $-\boldsymbol{\omega}_i$ and the normal of $\boldsymbol{p'}$ ($\boldsymbol{n}_{\boldsymbol{p'}}$). Now using equation (17), (18), (19) and (20) we arrive at the area form of the rendering equation:

$$L(\boldsymbol{p'} \to \boldsymbol{p}) = L_e(\boldsymbol{p'} \to \boldsymbol{p}) + \int_A f_r(\boldsymbol{p''} \to \boldsymbol{p'} \to \boldsymbol{p}) L(\boldsymbol{p''} \to \boldsymbol{p'}) G(\boldsymbol{p''} \leftrightarrow \boldsymbol{p'})\ dA(\boldsymbol{p''}) \tag{21}$$

This equation above lets us think about more interesting light transport techniques by converting it further to a formulation known as the **path integral formulation** of light transport which will be discussed later. For now, we need a way to approximate our unwieldy reflectance integral and for that we move our attention towards **monte carlo integration**.

4.4 Monte Carlo Integration

The reflectance integral in the rendering equation is given by:

$$L_r(\boldsymbol{p}, \boldsymbol{\omega}_r) = \int_\Omega f_r(\boldsymbol{\omega}_i, \boldsymbol{p}, \boldsymbol{\omega}_o)\ L_i(\boldsymbol{p}, \boldsymbol{\omega}_i)\ cos\theta_i\ d\boldsymbol{\omega}_i \tag{22}$$

This integral is not feasible to compute in closed form since it can include various complex BRDF functions for various objects in our scene while also being a product of incident light. The integral is also recursive because of the nature of the rendering equation. Provided that we have a complex function that is difficult to integrate in closed form, we can always sample the function in its domain. And so given an integral:

$$I = \int_a^b f(x)dx \tag{23}$$

the steps to compute the monte carlo estimate of the integral are:

1. Pick random samples in the function's domain
2. Sum up those function values obtained at those samples
3. Take average of that summation
4. Multiply the value obtained by the volume of the domain

Mathematically,

$$I = \lim_{n\to\infty} \frac{(b-a)}{n} \sum_{i=1}^{n} f(X_i) \tag{24}$$

where, X_i are the random samples. This formulation states that as the number of samples tends to infinity we will eventually converge to our solution. The problem here is that while ray tracing, each of these sample is quite expensive to compute. Each sample is basically a ray that we shoot towards the scene and our monte carlo formulation states that increasing these samples would bring us closer to our desired image. These rays however, intersect with millions or possibly billions of triangles in our scene and so just increasing these samples is not the best solution. What we want is to trace as few rays as possible while also approximating our integral as exactly as possible since errors are perceived as noise in our resultant image. In order to improve on our naive approach we need to introduce some probability concepts starting with **probability density function** or **PDF**.

4.4.1 Probability Density Function (PDF)

In probability, we have continuous random variables X which are variables that can take infinitely many values. A probability density function (PDF) then describes the probabilities in the domain of this random variable. A very simple example of probability distribution function (PDF for discrete random variables) is that of an unbiased coin toss. An unbiased coin toss can take up two values of heads and tails, each with a probability of 0.5. Note, that this function's values also sum up to 1 which is one of the properties of a PDF. One other property is that all probability values should be positive i.e $p(x) \geq 0$. Also, the probability that the random variable is in a specified domain is always 1 i.e $\int_a^b p(x)dx = 1$.

4.4.2 Cumulative Distribution Function (CDF)

One other important function to discuss here is the **Cumulative Distribution Function** or **CDF** which is required to cumulate/sum probabilities from a PDF in a given domain. What this means is that a CDF at a given point in our PDF is the summation of all values in the PDF from 0 to that point. Mathematically:

$$P(x) = \int_0^x p(x)dx \tag{25}$$

And CDF in a given domain, from a to b is the summation of all values in the PDF in that domain. Mathematically:

$$P(a \leq X \leq b) = \int_a^b p(x)dx = P(b) - P(a) \tag{26}$$

A CDF basically tells us about the probability of the events occurring in a given domain. This helps us to generate values according to our desired PDF.

4.4.3 Expected Value

The **expected value** is a measure of what value does a random variable take on average? It is the arithmetic mean of independently selected outcomes of a random variable. Mathematically:

$$E(X) = \sum_{i=1}^{n} p_i x_i \tag{27}$$

where, p_i is the probability of the outcome and x_i is the value of the outcome. Considering an example of a fair six-sided die, the probability of the die landing on any number from 1 to 6 is $\frac{1}{6}$ each. The expected value is then calculated as:

$$E(X) = 1 \cdot (\frac{1}{6}) + 2 \cdot (\frac{1}{6}) + 3 \cdot (\frac{1}{6}) + 4 \cdot (\frac{1}{6}) + 5 \cdot (\frac{1}{6}) + 6 \cdot (\frac{1}{6}) = 3.5 \tag{28}$$

What this result tells us is that, as you roll the die a repeated number of times, the value that we get would always be closer to 3.5 and as the number of trials are increased, the value would slowly converge to this value of 3.5. For continuous random variables, the expected value is the mean μ of the probability density function:

$$E(X) = \int_\Omega p(x)x \; dx \tag{29}$$

In terms of rendering, the expected value of our function that estimates the reflectance integral converges to the solution of the reflectance integral.

4.4.4 Variance

Variance is a measure of how far our random samples are from average, on average. This might seem a bit confusing but intuitively variance tells us how unpredictable our samples are, as they are deviating from the average. So for a simple example of a coin toss but this time considering a biased coin which always turns up heads. It is quite obvious that its variance would be 0 because on *average* it is always giving us the same exact value so our outcomes are *predictable*. Mathematically variance is defined as:

$$V(X) = E[(X - E[X])^2] \tag{30}$$

In terms of rendering, high variance would mean that our samples are deviating from approximating the original reflectance integral and this shows up as noise in our images. More samples would help in decreasing noise but there is a point where we have diminishing returns which means that increasing the samples no longer help us in decreasing visible amount of noise. So our goal here is to decrease variance by taking more samples in areas where our reflectance integral is large thereby, efficiently using our samples.

4.4.5 Central Limit Theorem (CLT)

It is important to also mention the **Central Limit Theorem (CLT)** here. CLT states that when taking the mean of $N \to \infty$ independently and identically distributed random variables X with defined expectation μ and finite variance σ^2, the resulting random variable X_N will tend towards a normal distribution:

$$\lim_{N\to\infty} \frac{1}{N} \sum_{i=1}^{N} X_i = \hat{X}_N \sim \mathcal{N}(\mu, \frac{\sigma^2}{N}) \tag{31}$$

It is important to note that knowing this theorem can increase the possibilities of analyzing scenes that are rendered using a monte carlo estimator.

4.4.6 Importance Sampling

So far, we have been taking our samples randomly in order to approximate our reflectance integral but we can however pick samples from a non-uniform distribution, or in other words, picking more samples in one area of our function than other. This is often called **biasing** since we are being biased for different areas of our integral. We are trying to sample more in areas where we can improve the value of our integral more and less where the samples do not contribute much. So we weight the contribution of each sample by how likely we were to pick it. The formula for our estimator becomes:

$$I = \frac{1}{n} \sum_{i=1}^{n} \frac{f(X_i)}{p(X_i)} \tag{32}$$

where $f(X_i)$ is again our function value at the random sample X_i but now we weight each of our sample with a PDF at that sample.

One example of importance sampling in rendering is **BRDF sampling**. And as the name suggests, BRDF sampling is a type of sampling which takes the BRDF of the material into account. This means that we will try to sample more where more light is reflected on a specific material. For glossy surfaces, this means that we want to sample more at specular regions of the material as that is what gives glossy surfaces their *plastic like* appearance.

4.4.7 Multiple Importance Sampling (MIS)

We have already discussed importance sampling but what if sampling from a single distribution still doesn't estimate the final integral properly. In this case, we try to importance sample multiple distributions, and so we arrive at **multiple importance sampling** [40][21]. The idea is simple yet very effective. We generate samples from multiple distributions, each suited for different parts of our integrand and then combine these samples to get an overall estimate. The resulting estimator is as follows:

$$I_N^{MIS} = \sum_{i=1}^{n} \frac{1}{n_i} \sum_{j=1}^{n_i} w_i(X_{i,j}) \frac{f(X_{i,j})}{p_i(X_{i,j})} \tag{33}$$

where w_i is a weighting function associated with the pdf p_i, and n_i is the number of samples taken from p_i. The estimator is unbiased as long as the weighting functions w_i, referred to as the *weighting heuristic*, fulfills the following conditions:

1. $\sum_{i=1}^{N} w_i(x) = 1$ whenever $f(x) \neq 0$
2. $w_i(x) = 0$ whenever $p_i(x) = 0$

5 Implementation

5.1 Nori - An educational ray tracer

The renderer, Nori [25], used in this thesis is provided as an educational resource from EPFL. Nori is an educational ray tracer which is used by different institutes (including EPFL, ETH Zurich, Cornell and many others) as a framework to teach path tracing to their students. But we are not restricted to just path tracing when working on Nori. In fact, we can implement a number of other rendering algorithms since it gives you the skeleton of the inner workings of a ray tracer. The base code of Nori provides many features that would be tedious to implement from scratch. This includes:

1. A simple GUI to watch images as they render
2. An XML-based scene file loader
3. Basic point/vector/normal/ray/bounding box classes
4. A pseudorandom number generator (PCG32)
5. Support for saving output as OpenEXR files
6. A loader for Wavefront OBJ files
7. Ray-triangle intersection
8. Code for multi-threaded rendering
9. Image reconstruction filters
10. Statistical tests to verify sampling code
11. A graphical visualization tool that can be used to inspect and validate sampling code.

So the above code only gives us the skeleton code necessary to get started with building our own path tracing implementation while the main bits of the code are empty. The implementation that we will add includes:

1. Adding an acceleration structure to speed up our intersection tests
2. Monte Carlo Sampling
3. Point Lights, Ambient Occlusion, Area Lights
4. Direct Illumination Integrator
5. Whitted Style Integrator
6. Path Tracing

5.2 Acceleration Data Structure

Acceleration data structures play an important role in any ray tracer, reducing the amount of intersections of each ray with objects in our scene. Mainly, there are two broad categories of acceleration structures namely, spatial subdivision and object subdivision. Spatial subdivision algorithms divide 3D space into regions and record which primitives overlap which regions. On the other hand, object subdivision algorithms progressively break the objects in the scene into smaller sets of objects. We added BVH which is an example of object subdivision acceleration structure.

5.2.1 Concept

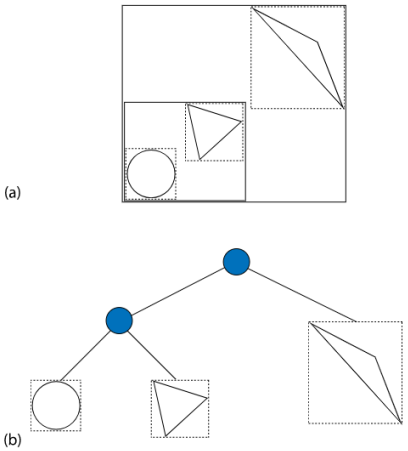


Figure 7: A hierarchy of bounding volumes of a scene consisting of 3 primitives. [32]

The idea in BVH is to partition complex objects into a hierarchy of simpler objects that are easier to test. If we send a ray into the scene shown in the figure above, our first test would be with the scene **axis aligned bounding box** (AABB) itself (a), and if our ray does intersect with the scene AABB or the root node, we can then move further and test similarly with the two AABBs in the left and right node. Now there's a chance that our ray hits the left node again where we would test the same ray again but this time only with the primitives' AABBs in the left node. And if our ray does not hit the right node, then that would mean that we also do not intersect with any of the primitives in the right node or for more complex scenes, any of the further nodes in the right node. This results in a much more efficient way to test if we have an intersection with complex models in our scene. This way of dividing objects is known as **object subdivision**.

5.2.2 Implementation

We do have an idea of how to build the hierarchy or tree of primitives in our scene but there are still some important questions to answer. One of the most important part in a BVH is the **splitting criteria**. As we go through all the primitives in our scene, we need a criteria that decides how to partition our primitives in the left and right node. This is important because a bad criteria can result in a bad tree where our primitives in both left and right node could overlap. This would result in more ray-primitive intersection tests when the tree is traversed and thereby resulting in a bad implementation of BVH. The splitting criteria which we have used to split primitives is called the **centroid method**. The method starts with computing the centroid/midpoint of all primitives' AABBs in a node. This is done by first calculating the AABB of a primitive, and then calculating centroid $\boldsymbol{c}$ as:

$$\boldsymbol{c} = (\boldsymbol{primitive}_{\boldsymbol{AABB}_{\boldsymbol{min}}} * 0.5) + (\boldsymbol{primitive}_{\boldsymbol{AABB}_{\boldsymbol{max}}} * 0.5) \tag{34}$$

As we are computing centroids, we can get the minimum and maximum for all the centroids in a node. This minimum and maximum of all centroids is then used to compute an AABB which represents the AABB based on centroids of all the primitives in this node. Once we have this AABB, we then compute the diagonal $\boldsymbol{d}$ of this AABB as:

$$\boldsymbol{d} = \boldsymbol{AABB}_{\boldsymbol{max}} - \boldsymbol{AABB}_{\boldsymbol{min}} \tag{35}$$

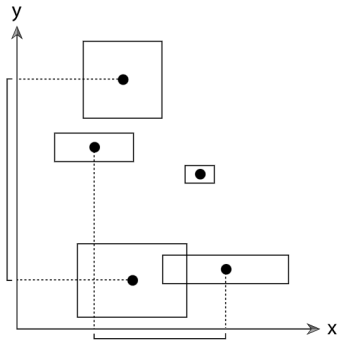


Figure 8: Splitting primitves in a scene based on centroids of their bounding volumes. Note that here, splitting in y-axis is the best way to split primitives.[32]

This diagonal $\boldsymbol{d}$ can then be used to find the maximum extent in a given axis. What this means is that, we check the x, y and z value of this diagonal $\boldsymbol{d}$ and the one which is the highest among the three is the axis we will use to split our primitives. Intuitively, what we did is chose the axis which results in the least overlap of all AABBs in a given node.

In the example in the figure above, this axis is y for that specific scene. In this way we end up with a tree which is using some of the information we have in our scene to select our axis. We still haven't decided the value which would partition our primitives. That part is quite simple. We just need to use the centroid AABB $\boldsymbol{c}$ we calculated previously, and calculate the midpoint of just the splitting axis we computed. So if the best splitting axis was y we compute this split value s as:

$$s = (c_{min_y} * 0.5) + (c_{max_y} * 0.5) \tag{36}$$

Ofcourse, if the best axis was x or z in another node, we need to use that axis in the above equation. In the figure above, this value we just computed is the midpoint of the extent in y so we end up with

2 primitives in the left node and 3 primitives in the right node. There is one slight issue with this approach. If all of the centroid points are at the same position (the centroid bounds have zero volume), we need to stop recursion and introduce a leaf node there. The condition is to check if the centroid AABB $\boldsymbol{c}$'s minimum value at the splitting axis is equal to it's maximum value at the splitting axis. For splitting axis y this condition is:

$$c_{min_y} == c_{max_y} \tag{37}$$

If the condition above is true for any splitting axis, we stop recursion and place first half of the primitives in left node and second half of the primtives in the right node. This wraps up our implementation of an acceleration structure. The BVH implemented is fast enough for relatively complex scenes. For example, the ajax bust which contains about 500k triangles renders in 8.2s for an image that visualizes normals.

5.3 Monte Carlo Sampling

We can now move on to start implementing sampling techniques. As previously discussed, in monte carlo integration, which is used to solve our unwieldy rendering equation, the very first step is to be able to pick random samples in our function's domain. This is the part where we will implement different sampling techniques for different distributions. The **inversion method** is an important technique to mention here which enables us to map a uniform distribution to a specific goal distribution. The steps involved in the inversion method are:

1. Create a PDF of the goal distribution
2. Get the CDF of that PDF by integrating it
3. Compute the inverse of that CDF
4. Map the uniformly distributed value to the goal distribution by using the inverse function obtained above

We will not be deriving each transformation of distributions here but only mentioning how a sample can be transformed from one distribution to another.

5.3.1 Transforming between Distributions

The mappings to be implemented in Nori that would enable us to warp our uniformly distributed point on a unit square to other distributions are:

1. **Square to Tent Distribution**
   A tent distribution can be used to map samples from a unit square to a unit triangle. Given that our function is:
   $$p(t) = \left\{ \begin{array}{ll} 1 - |t|, & -1 \leq t \leq 1 \\ 0, & \text{otherwise} \end{array} \right\} \tag{38}$$
   which represents a distribution graphed as:

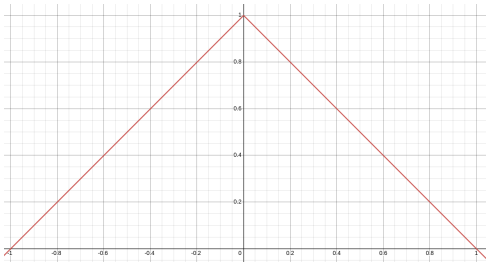


Figure 9: Tent distribution from -1 to 1.

the warped sample from unit square with sample points ξ_1 and ξ_2, to the distribution is obtained as:

$$u = 1 - \sqrt{1 - \xi_1} \tag{39}$$

$$v = \xi_2 \sqrt{1 - \xi_1} \tag{40}$$

The warped point on the triangle is:

$$\boldsymbol{p} = u(2, 0) + v(1, 1) - (1, 0) \tag{41}$$

We also need a way to test if a point is inside a triangle which would be the *pdf* of this tent distribution. We would return 1 if the point is inside triangle and 0 otherwise. One of the many ways to check if a point $\boldsymbol{p}$ is inside a triangle t_{ABC} is to check on which side of the half plane created by the edges of the triangle, the point lies in.

2. **Square to Uniform Disk**
Mapping samples from a unit square to unit disk is similar to transforming uniformly generated samples to a unit circle. The mapping used in our implementation is a concentric mapping which was also discussed in [32]. Traditional mapping from a unit square to unit disk have a problem of distorted areas on the disk i.e areas on the unit square are elongated when mapped to a disk.

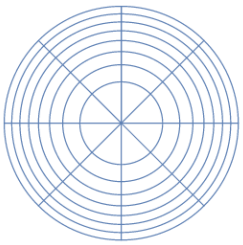


Figure 10: Traditional mapping of unit square to unit disk. Each section of the disk here has equal area and represents $\frac{1}{8}$ of the unit square of uniform random samples in each direction.

A better mapping takes points in the square $[-1, 1]^2$ to the unit disk by uniformly mapping concentric squares to concentric circles.

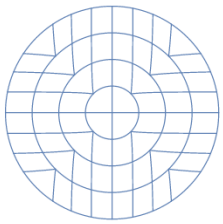


Figure 11: Concentric mapping maps squares to circles, giving a less distorted mapping.

In this mapping we essentially turn wedges of square to slices of disk. The mapping is done as follows:

$$\boldsymbol{u} = 2(\xi_1, \xi_2) - (1, 1) \tag{42}$$

$$\left\{ \begin{array}{ll} r = u_x, & \theta = \frac{\pi}{4}\frac{u_y}{u_x}, \quad |u_x| > |u_y| \\ r = u_y, & \theta = \frac{\pi}{2} - \frac{\pi}{4}\frac{u_x}{u_y}, \quad \text{otherwise} \end{array} \right\} \tag{43}$$

The r and θ are then the points that are warped to unit disk in a concentric manner:

$$\boldsymbol{p} = r(cos\theta, sin\theta) \tag{44}$$

Now we just need a way to test if a point lies on a circle. That is easy, we just check if the following condition holds given a point $\boldsymbol{p}$ on a unit disk:

$$p_x^2 + p_y^2 <= 1 \tag{45}$$

3. **Sqaure to Uniform Sphere**
   This involves mapping uniformly distributed samples to a uniform sphere. The formula to get the x, y, z coordinates of the sphere from a uniform sample first involves getting the two angles ϕ and θ:

$$\theta = cos^{-1}(1 - 2\xi_1) \tag{46}$$

$$\phi = 2\pi\xi_2 \tag{47}$$

   After getting θ and ϕ, we calculate the warped x, y, z coordinates of the sphere as follows:

$$x = sin(\theta)cos(\phi) \tag{48}$$

$$y = sin(\theta)sin(\phi) \tag{49}$$

$$z = cos(\theta) \tag{50}$$

   To test if a give point $\boldsymbol{p}$ is on a sphere, the test is given as follows where ϵ is an error value typically with a value 0.0001:

$$|p_x^2 + p_y^2 + p_z^2 - 1| <= \epsilon \tag{51}$$

4. **Square to Uniform Hemisphere**
   We now move towards mapping uniform samples to a uniform hemisphere or half sphere. This is quite similar to sampling points a uniform sphere but we only need the angle ϕ this time:

$$\phi = 2\pi\xi_2 \tag{52}$$

   And we calculate the warped x, y, z coordinates on the hemisphere as:

$$x = cos(\phi)\sqrt{1 - (1 - \xi_1)^2} \tag{53}$$

$$y = sin(\phi)\sqrt{1-(1-\xi_1)^2} \tag{54}$$

$$z = 1 - \xi_1 \tag{55}$$

To test if a given point $\boldsymbol{p}$ is on a hemisphere we test if:

$$|p_x^2 + p_y^2 + p_z^2 - 1| <= \epsilon \quad and \quad p_z \geq 0 \tag{56}$$

5. **Square to Cosine Weighted Hemisphere**
Cosine weighted hemisphere sampling is similar to hemisphere sampling except the fact that here we try to generate more samples in the normal direction and less samples close to tangential angles. This is often used in rendering because when solving the reflectance integral, we know that radiance $\boldsymbol{L}$ has a larger effect on the irrdiance $\boldsymbol{E}$ or an area for smaller angles of θ. We again calculate ϕ as follows:

$$\phi = 2\pi\xi_2 \tag{57}$$

and then we calculate the warped sample as:

$$x = cos(\phi)\sqrt{\xi_1} \tag{58}$$

$$y = sin(\phi)\sqrt{\xi_1} \tag{59}$$

$$z = \sqrt{1-\xi_1} \tag{60}$$

The test remains the same as uniform hemisphere sampling.

6. **Square to Beckmann Distribution**
Beckmann distribution is used to model pdf of normals on rough surfaces. This is applicable for when we want to implement the microfacet BRDF. A beckmann distribution for different values of α looks like this:

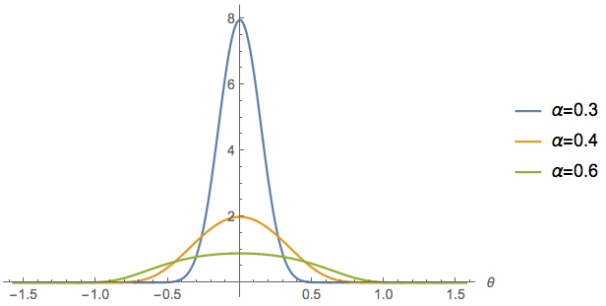


Figure 12: Beckmann distribution for different values of α [25].

We need to calculate two angles ϕ and θ given an α value. They are computed as follows:

$$\phi = 2\pi\xi_1 \tag{61}$$

$$\theta = tan^{-1}(\sqrt{-\alpha^2 log(1-\xi_2)}) \tag{62}$$

and the warped sample is calculated as:

$$x = sin(\theta)cos(\phi) \tag{63}$$

$$y = sin(\theta)sin(\phi) \tag{64}$$

$$z = cos(\theta) \tag{65}$$

But the vector above needs to be normalized. The pdf of the beckmann distribution will then take this vector $\boldsymbol{p}$ and α and is calculated as follows:

$$tan\theta = tan(acos(p_z)) \tag{66}$$

$$pdf = \frac{1}{2\pi} * \frac{2e^{\frac{-tan\theta^2}{\alpha^2}}}{\alpha p_z^3} \tag{67}$$

5.3.2 Point Lights

5.3.2.1 Concept

A **point light** is one of the most basic means of local illumination in a ray tracer. As the name suggests, a point light is a point in 3D space which emits light. A light source emits power or flux denoted by Φ. So our point light emits flux and has a position $\boldsymbol{x}$. This is sufficient to implement basic local illumination in our scenes as we know that the point emits a specific amount of flux which can be lower or higher depending on its value which will in turn also affect how the objects in our scene gets illuminated. And with its position, we also know how far the point is from objects in our scenes.

5.3.2.2 Implementation

So lets define a point light in our scene with flux Φ and position $\boldsymbol{x_1}$. We have a point $\boldsymbol{x}$ on a triangle in our scene with normal $\boldsymbol{n}$. The distance from the point light to the point to be shaded is given by r.

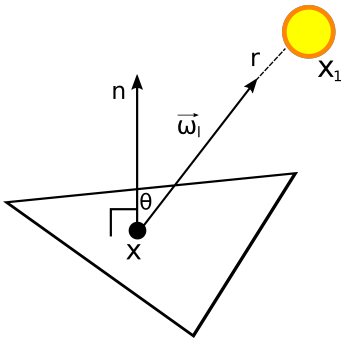


Figure 13: A point light $\boldsymbol{x_1}$ and a point $\boldsymbol{x}$ to be shaded on a triangle.

So we have to measure how much light hits this point on the triangle. More specifically, we are interested in calculating the irradiance which is the radiant flux per unit area. The irradiance on a point or the density of photons arriving at a point will be much higher when the light emitting the photons is closer and it will be more spread out when the point light is further away. Mathematically, the irradiance reaching that point is given by:

$$E = \frac{\Phi}{4\pi r^2} cos\theta = \frac{\Phi}{4\pi r^2}(\boldsymbol{n} \cdot \boldsymbol{\omega_l}) \tag{68}$$

The cosine term or the dot product between the normal $\boldsymbol{n}$ and the vector $\boldsymbol{\omega_l}$ is known as the **foreshortening term** and handles the amount of light arriving at the point based on the angle the light is at. A pseudocode of the integrator is as follows:

Require: scene S, sampler s, ray r
Ensure: radiance L_i
function POINTLIGHTINTEGRATOR(S, s, r)
if S.rayIntersect(r, intersection) is false **then**
return Color(0, 0, 0)
l ← lightp - intersection.position
dist ← = l.sqauredNorm()
l.normalize() ▷ compute shadow ray
if S.rayIntersect(Ray(intersection.position, l)) is true **then**
return Color(0, 0, 0)
▷ compute illumination
costheta ← l.dot(intersection.normal)
$L_i \leftarrow \frac{e}{4\times\pi^2} \times max(0, costheta) \times \frac{1}{dist}$

return L_i

5.3.3 Ambient Occlusion

5.3.3.1 Concept

Next, we will add an **ambient occlusion** integrator. A rendering technique that assumes that a diffuse surface receives uniform illumination from all directions and that visibility is the only effect that matters. Some surface positions will receive less light than others since they are occluded, hence they will look darker. This will also make use of the cosine weighted hemisphere sampling we implemented previously. Starting with the reflectance integral:

$$L_r(\boldsymbol{p}, \boldsymbol{\omega}_r) = \int_\Omega f_r(\boldsymbol{\omega}_i, \boldsymbol{p}, \boldsymbol{\omega}_r)\ L_i(\boldsymbol{p}, \boldsymbol{\omega}_i)\ cos\theta_i\ d\boldsymbol{\omega}_i \tag{69}$$

Since we have diffuse objects, the BRDF is (9). We also have ambient white sky which means the incident radiance, independent of the direction will always be white $(1, 1, 1)$. This simplifies equation (69) above to:

$$L_r(\boldsymbol{p}) = \frac{\rho}{\pi} \int_\Omega V(\boldsymbol{p}, \boldsymbol{\omega}_i) cos\theta_i\ d\boldsymbol{\omega}_i \tag{70}$$

where V is the visibility function. Our equation is still an integral though and we would like to approximate it by using a sum.

5.3.3.2 Implementation

So in order to approximate equation (70) using a sum, we can rewrite it as:

$$L_r(\boldsymbol{p}) \approx \frac{\rho}{\pi} \sum_{k=1}^{N} \frac{V(\boldsymbol{p}, \boldsymbol{\omega}_{i,k}) cos\theta_{i,k}}{p(\boldsymbol{\omega}_{i,k})} \tag{71}$$

where N is the number of samples used to approximate our integral, and $p(\boldsymbol{\omega}_{i,k})$ is the pdf we use to importance sample our approximation. The closer this pdf is to our actual function, the less variance (noise) is in our rendered image. This is where our cosine weighted hemisphere sampling would play an important role. We use $p(\boldsymbol{\omega}_{i,k}) = \frac{cos\theta_{i,k}}{\pi}$ in equation (71) which gives us:

$$L_r(\boldsymbol{p}) \approx \frac{\rho}{N} \sum_{k=1}^{N} V(\boldsymbol{p}, \boldsymbol{\omega}_{i,k}) \tag{72}$$

and this is what the ambient occlusion integrator looks like with cosine weighted sampling. A pseudocode of the integrator is as follows:

Require: scene S, sampler s, ray r
Ensure: color L_i
  function AOINTEGRATOR(S, s, r)
    if S.rayIntersect(r, intersection) is false **then**
      return Color(0, 0, 0)
    ▷ get sampled direction
    sampledDir = squareToCosineHemisphere(s.2D())
    if S.rayIntersect(Ray(intersection.position, sampledDir)) is true **then**
      return Color(0, 0, 0)
    else
      return Color(1, 1, 1)

Ambient occlusion is a very powerful yet an efficient method to create renderings that already show how ray tracing is better than rasterization. Soft shadows which is often difficult to implement in rasterization based systems can already be observed when using ambient occlusion.

5.4 Materials

This is a good point to discuss how to model materials in a physically based path tracer. As discussed previously all materials have a Bidirectional Reflectance Distribution Function or BRDF. We will implement 3 different kinds of BRDFs namely, diffuse, dielectric and microfacet. All BRDFs have a function to *evaluate* which takes in incoming and outgoing directions and returns the evaluated color, a function to calculate the *pdf* which will tell us how much light was reflected in the form of a scalar value, and a function to *sample* the BRDF, which as the name suggests allows us to sample a point on the BRDF. We will start our discussion with the diffuse material.

5.4.1 Diffuse

A BRDF for a diffuse material is relatively simple. This material reflects light equally in all directions so the evaluate function for a diffuse material is given by:

$$fr_d = \frac{\boldsymbol{\rho}}{\pi} \tag{73}$$

where, $\boldsymbol{\rho}$ is the albedo or the reflectance coefficient of the material set by us. The pdf of a diffuse BRDF is given by:

$$d_{pdf} = \frac{\boldsymbol{n} \cdot \boldsymbol{\omega_o}}{\pi} \tag{74}$$

where $\boldsymbol{n}$ is the normal of the point being shaded on the diffuse material and $\boldsymbol{\omega_o}$ is the outgoing light direction from the material which is given by the cosine weighted hemisphere sampling when sampling the BRDF. The sample function itself would return the albedo $\boldsymbol{\rho}$.

5.4.2 Dielectric

A dielectric material is named so because it contains all types of materials that are poor conductors of electricity. Two common materials include glass and water. It takes into account reflection as well as refraction i.e how the light is reflected and how it is bent when it enters the material. In order to know when light is reflected or refracted, we have to make use of the *fresnel equations* [20]. The fresnel equations tell us how much light is reflected vs how much of it is refracted given an incoming light direction. For dielectrics, light is either polarized parallel or perpendicular to the plane of refraction:

$$\rho_{\parallel} = \frac{\eta_i cos\theta_1 - \eta_e cos\theta_2}{\eta_i cos\theta_1 + \eta_e cos\theta_2} \tag{75}$$

$$\rho_{\perp} = \frac{\eta_e cos\theta_1 - \eta_i cos\theta_2}{\eta_e cos\theta_1 + \eta_i cos\theta_2} \tag{76}$$

The fresnel constant for light reflected F_r and light refracted F_t are computed as:

$$F_r = \frac{1}{2}(\rho_{\parallel}^2 + \rho_{\perp}^2) \tag{77}$$

$$F_t = 1 - F_r \tag{78}$$

Our fresnel function takes the cosine of the angle between the normal and the incoming direction, the interior refractive index η_i as well as the exterior refractive index η_e of the material. It then returns a fresnel constant (a scalar) that is then compared with a random sample and decided if it should be a reflection:

$$\boldsymbol{\omega}_o = -\boldsymbol{\omega}_i \tag{79}$$

or refraction according to snell's law [18]:

$$\boldsymbol{\omega}_o = -\frac{\eta_e}{\eta_i}(\boldsymbol{\omega}_i - (\boldsymbol{\omega}_i \cdot \boldsymbol{n})\boldsymbol{n}) - \boldsymbol{n}\sqrt{1 - (\frac{\eta_e}{\eta_i})^2(1 - (\boldsymbol{\omega}_i \cdot \boldsymbol{n})^2)} \tag{80}$$

This is how the dielectric BRDF's sample function would look like. It is to be noted that both the evaluation as well as pdf of a dielectric return 0 since it is a discrete BRDF i.e reflects light in only one direction. There is another more appropriate term for a dielectric BRDF which is the **Bidirectional Scattering Distribution Function** or **BSDF** because our BRDF not just reflects light (light rays going in known directions) but also transmit them (light rays being refracted) and overall it is telling us how light is scattered.

5.4.3 Microfacet

The microfacet BSDF is based on the microfacet theory introduced by Blinn [7] and Cook and Torrance [15]. The motivation behind the model was to better understand and model reflection of light from rough surfaces. The microfacet theory models the surface of a material as tiny surfaces (microfacets) with varying slope and height [8]. We will implement the GGX microfacet model introduced by Walter et al. [41]. The microfacet BSDF evaluation function is defined as follows:

$$f_r(\boldsymbol{\omega}_i, \boldsymbol{\omega}_o) = \frac{k_d}{\pi} + k_s \frac{D(\boldsymbol{\omega}_h)F((\boldsymbol{\omega}_h \cdot \boldsymbol{\omega}_i), \eta_e, \eta_i)G(\boldsymbol{\omega}_i, \boldsymbol{\omega}_o, \boldsymbol{\omega}_h)}{4\ cos\theta_i\ cos\theta_o\ cos\theta_h} \tag{81}$$

where $\boldsymbol{\omega}_h$ is,

$$\boldsymbol{\omega}_h = \frac{(\boldsymbol{\omega}_i + \boldsymbol{\omega}_o)}{||\boldsymbol{\omega}_i + \boldsymbol{\omega}_o||^2} \tag{82}$$

$k_d \in [0, 1]^3$ is RGB diffuse reflection coefficient. This is similar to setting an albedo value in a diffuse BSDF, only this time it will alow us to set color to our microfacet material. $k_s = 1 - max(k_d)$ is the specular reflection coefficient, F is the fresnel reflection coefficient which is the same function we described when discussing dielectric material, η_e is the exterior index of refraction, η_i is the interior index of refraction and D is a normal distribution function. D actually tells us how much these microfacets vary. For D we use the beckmann distribution which is defined as:

$$D_{beckmann} = \frac{e^{\frac{-tan^2\boldsymbol{\omega}_h}{\alpha^2}}}{\pi\alpha^2 cos^4\boldsymbol{\omega}_h} \tag{83}$$

G is called the shadowing-masking function. It is named so, because shadowing occurs when incident light is blocked by other microfacet and masking occurs when reflected light is blocked by another microfacet. It is calculated as:

$$G(\boldsymbol{\omega}_i, \boldsymbol{\omega}_o, \boldsymbol{\omega}_h) = G_1(\boldsymbol{\omega}_i, \boldsymbol{\omega}_h) G_1(\boldsymbol{\omega}_o, \boldsymbol{\omega}_h) \tag{84}$$

$$G_1(\boldsymbol{\omega}_v, \boldsymbol{\omega}_h) = \chi^+ \left(\frac{\boldsymbol{\omega}_v \cdot \boldsymbol{\omega}_h}{\boldsymbol{\omega}_v \cdot \boldsymbol{n}} \right) \begin{cases} \frac{3.535b + 2.181b^2}{1 + 2.276b + 2.577b^2}, & b < 1.6 \\ 1, & otherwise \end{cases} \tag{85}$$

$$b = \alpha(tan\theta_v)^{-1}, \;\; \chi^+(c) = \begin{cases} 1, & c > 0 \\ 0, & c \leq 0 \end{cases} \tag{86}$$

where $\chi^+(c)$ is called the *positive characteristic function* and θ_v is the angle between the surface normal $\boldsymbol{n}$ and $\boldsymbol{\omega}_v$ passed to G_1. This wraps up our evaluation function for our microfacet BSDF. But we still need to be calculate the pdf of our BSDF. This is done by the following function:

$$M_{pdf} = (1 - k_s) \frac{cos\theta_o}{\pi} + k_s \; D(\omega_h) \; J_h \tag{87}$$

where $J_h = (4(\boldsymbol{\omega}_h \cdot \boldsymbol{\omega}_o))^{-1}$ is the Jacobian of the half direction mapping. Also, θ_o, which is the outgoing direction, is actually set when we sample the function. It is the cosine weighted hemisphere sampling when our random sample is greater than k_s otherwise we first get $\boldsymbol{\omega}_h$ using beckmann distribution and then compute θ_o as:

$$\theta_o = ||((2 * (\boldsymbol{\omega}_h \cdot \boldsymbol{\omega}_i) * \boldsymbol{\omega}_h) - \boldsymbol{\omega}_i)|| \tag{88}$$

And the sample function itself would return:

$$\boldsymbol{M}_s = f_r(\boldsymbol{\omega}_i, \boldsymbol{\omega}_o) * \frac{cos\theta_o}{M_{pdf}} \tag{89}$$

5.5 Distribution and Whitted-style Ray Tracing

In order to be able to realize our first ray traced image, we need a way to convert objects in our scene to emissive materials. Emissive materials are materials that emit light. The goal here is to transform any geometric object to a light source (emitter). For this purpose we need to be able to sample triangles on a mesh. We have to implement a method that gets the sampled position $\boldsymbol{p}$ on the mesh, the interpolated surface normal $\boldsymbol{n}$ and the probability density of the sample which is the reciprocal of the surface area of the entire mesh.

5.5.1 Area lights

The first step here is to be able to sample any mesh uniformly. For this purpose, we first need to calculate the discrete probability density (pdf) of the mesh we are working with. We go through each triangle in a mesh one by one and get the surface area of each of the triangles. After getting the surface area, we normalize the discrete pdf. Next, we sample a triangle uniformly from this normalized discrete pdf.
Now we need to sample a barycentric coordinate on the sampled triangle. We do this as given in equation (39) and (39).

Get the sampled position $\boldsymbol{p}$ on this triangle by:

$$\boldsymbol{p} = u\boldsymbol{v_1} + v\boldsymbol{v_2} + (1 - u - v)\boldsymbol{v_3} \tag{90}$$

where $\boldsymbol{v_1}$, $\boldsymbol{v_2}$ and $\boldsymbol{v_3}$ are the vertices of the sampled triangle. If a mesh has per vertex normals $\boldsymbol{n_1}$, $\boldsymbol{n_2}$, $\boldsymbol{n_3}$, we can get the interpolated normal $\boldsymbol{n}$ as:

$$\boldsymbol{n} = u\boldsymbol{n_1} + v\boldsymbol{n_2} + (1-u-v)\boldsymbol{n_3} \tag{91}$$

Note, that the above vector $\boldsymbol{n}$ should also be normalized. If we don't have per vertex normals, we just compute the edges $\boldsymbol{e_1}$ and $\boldsymbol{e_2}$. These are calculated as $\boldsymbol{e_1} = \boldsymbol{v}_2 - \boldsymbol{v_1}$ and $\boldsymbol{e_2} = \boldsymbol{v}_3 - \boldsymbol{v_1}$. Next, we take the cross product of these two edges and then normalize it to get the normal. The probability density as already stated is the reciprocal of the surface area of the entire mesh. Our area lights would take a query record as an argument which includes the position to be shaded $\boldsymbol{p}$, the normal of the shading position $\boldsymbol{n}$, the position of the light $\boldsymbol{l_p}$ and the normal of the light position $\boldsymbol{l_n}$. The radiance reaching a point from the light position is computed as follows:

$$\boldsymbol{\Phi}_r = \boldsymbol{\Phi}(\|\boldsymbol{l_p} - \boldsymbol{p}\| \cdot \boldsymbol{n}) \tag{92}$$

where $\boldsymbol{\Phi}$ is the radiance of our area light which is uniform for all sampled points and is specified by us.

5.5.2 Direct Illumination

Next we implement our very first, direct illumination integrator. By **direct illumination** we mean that for now, we will only illuminate surfaces by first hit (bounce) of the light and not further. We will again use monte carlo integration to approximate our reflectance integral.

$$L_r(\boldsymbol{p}, \boldsymbol{\omega}_r) = \int_\Omega f_r(\boldsymbol{\omega}_i, \boldsymbol{p}, \boldsymbol{\omega}_o)\ L_i(\boldsymbol{p}, \boldsymbol{\omega}_i)\ cos\theta_i\ d\boldsymbol{\omega}_i \tag{93}$$

Since we are only dealing with direct illumination for now, this means that $L_i(\boldsymbol{p}, \boldsymbol{\omega}_i)$ would be zero everywhere except for rays that hit an area light source. Instead of uniformly sampling directions on the hemisphere (which is also extremely inefficient), we will directly sample light sources and check if they are visible as seen from position $\boldsymbol{p}$. This means that we integrate over the light sources $\mathcal{L}$ instead of the hemisphere Ω.

$$L_r(\boldsymbol{p}, \boldsymbol{\omega}_r) = \int_\mathcal{L} f_r(\boldsymbol{p}, \boldsymbol{p} \to \boldsymbol{p'}, \boldsymbol{\omega}_r) L_e(\boldsymbol{p'}, \boldsymbol{p'} \to \boldsymbol{p}) d\boldsymbol{p} \tag{94}$$

where $\boldsymbol{p} \to \boldsymbol{p'}$ is the normalized direction from $\boldsymbol{p}$ to $\boldsymbol{p'}$. Here $d\boldsymbol{p}$ is incorrect though since we changed our integral from solid angles to positions. The change we need is the addition of the geometric term G.

$$G(\boldsymbol{p} \leftrightarrow \boldsymbol{p'}) := V(\boldsymbol{p} \leftrightarrow \boldsymbol{p'}) \frac{|\boldsymbol{n}_p \cdot (\boldsymbol{p} \to \boldsymbol{p'})| \cdot |\boldsymbol{n}_{p'} \cdot (\boldsymbol{p'} \to \boldsymbol{p})|}{||\boldsymbol{p} - \boldsymbol{p'}||^2} \tag{95}$$

The final form of the rendering equation now becomes:

$$L_r(\boldsymbol{p}, \boldsymbol{\omega}_r) = \int_\mathcal{L} f_r(\boldsymbol{p}, \boldsymbol{p} \to \boldsymbol{p'}, \boldsymbol{\omega}_r) G(\boldsymbol{p} \leftrightarrow \boldsymbol{p'}) L_e(\boldsymbol{p'}, \boldsymbol{p'} \to \boldsymbol{p}) d\boldsymbol{p} \tag{96}$$

We still need to approximate our integral somehow and we'll do it by using monte carlo integration:

$$L_r(\boldsymbol{p}, \boldsymbol{\omega}_r) \approx \frac{1}{N} \sum_{k=1}^{N} \frac{f_r(\boldsymbol{p}_k, \boldsymbol{p}_k \to \boldsymbol{p'}, \boldsymbol{\omega}_r) G(\boldsymbol{p}_k \leftrightarrow \boldsymbol{p'}) L_e(\boldsymbol{p'}, \boldsymbol{p'} \to \boldsymbol{p}_k)}{p_\mathcal{L}(\boldsymbol{p'})} \tag{97}$$

The above equation tells us that in order to get the radiance reaching a point $\boldsymbol{p}$ we need to get the BSDF value accompanied with the geometric term and the emitted light arriving from our light in our

scene [4]. All of this needs to be divided by the pdf of the light source. The pdf of the light source is given by:

$$p_{\mathcal{L}} = \frac{1}{light_{pdf}} \tag{98}$$

Here $light_{pdf}$ for a mesh is the normalized surface area of all the triangles in a mesh. We can plug the above pdf in our equation (91) and the final form of the direct illumination integrator looks like:

$$L_r(\boldsymbol{p}, \boldsymbol{\omega}_r) \approx \frac{light_{pdf}}{N} \sum_{k=1}^{N} f_r(\boldsymbol{p}_k, \boldsymbol{p}_k \rightarrow \boldsymbol{p'}, \boldsymbol{\omega}_r) G(\boldsymbol{p}_k \leftrightarrow \boldsymbol{p'}) L_e(\boldsymbol{p'}, \boldsymbol{p'} \rightarrow \boldsymbol{p}_k) \tag{99}$$

Note that this integrator is only for one area light in the scene. But similarly this can be computed for all area light sources in the scene.

5.5.3 Whitted Style Ray tracing

The integrator we implemented previously does not take into account multiple bounces of rays since it only computes light arriving directly at a position in our scene. But there can be objects in our scene which reflect light in specific directions such as a dielectric or a microfacet material. For such surfaces, it is important that when the ray is reflected, we recurse through the same algorithm again. So **whitted style ray tracing** involves checking if the ray hits a diffuse or specular material, for diffuse material we fall back to our previous implementation but for a specular material, we will generate a refracted direction producing a sampling weight c and a new direction $\boldsymbol{\omega}_r$. We then return the following radiance estimate:

$$L_i(c, \boldsymbol{\omega}_c) = \left\{ \begin{array}{ll} \frac{1}{0.95}\ c\ L_i(\boldsymbol{p}, \omega_r), & \text{if } \xi < 0.95 \\ 0, & \text{otherwise} \end{array} \right\} \tag{100}$$

where ξ is a random number. The psuedocode for both direct illumination and whitted style ray tracing combined is as follows:

Require: scene S, sampler s, ray r
Ensure: color L_i

function WHITTEDINTEGRATOR(S, s, r)
    $L_i \leftarrow 0$
    $L_{emitted} \leftarrow 0$
    intersection ← S.rayIntersect(r)
    if intersection is false **then**
        return Color(0, 0, 0)
    else
        if intersection.mesh.BSDF is diffuse **then** ▷ sample an emitter from the scene as em
                                                ▷ also get the pdf of the emitter in em_{pdf}
            $light_{pdf} \leftarrow 0$ ▷ calling sampleUniform also saves the pdf of light in $light_{pdf}$
            $light_p \leftarrow$ em.sampleUniform($light_{pdf}$)
            if intersection.mesh is also emitter **then**
                $L_{emitted} \leftarrow$ em.getRadiance()
                ▷ incidentLightRadiance computes light radiance as explained in area lights section
            $L_i \leftarrow L_i +$ incidentLightRadiance(em)
            $l \leftarrow light_p -$ intersection.position
            $dist \leftarrow l.squaredNorm().normalize()$
            $costheta \leftarrow -l.dot($intersection.normal$)$ ▷ evaluate BRDF
            $f \leftarrow$ intersection.mesh.BSDF.eval()
            $light_{pdf} \leftarrow \frac{light_{pdf} \times dist}{costheta}$

```
            light_pdf ← light_pdf / em_pdf
            return (L_emitted + L_i) / light_pdf × f
        else                                              ▷ sample specular BRDF
            sp ← intersection.mesh.BSDF.sample(s)
            if s.1D() < 0.95 and sp.x > 0 then
                return whittedIntegrator(S,s,intersection.wo) / (0.95 × sp)
            else
                return Color(0, 0, 0)
```

5.6 Path Tracing

5.6.1 Brute Force Path Tracer

The most naive version of path tracing relies on hitting light sources in the scene by chance instead of explicitly sampling points on them. So, we basically remove light sampling that we did earlier. We also account for *indirect illumination*. For this, we let the paths continue not only for specular surface interactions, but for all interactions. We will start this section by introducing the path integral formulation of the light transport equation. This form represents the rendering equation in a way that helps us think of light traveling in a scene in the form of paths. A path starts from the viewpoint and ends at the emitter or the light source in our scene. The different points that the light bounces along the way are called **vertices** of the path.

5.6.2 Path Integral Formulation of Light Transport Equation (LTE)

We have already defined the area form of the rendering equation and now we can express radiance as an integral over paths. One motivation for this formulation is that it allows us to measure an explicit integral over paths, as opposed to the recursive definition we have for our rendering equation. By substituting the right hand side of (21) into the $L(\boldsymbol{p''} \to \boldsymbol{p'})$ term inside the integral, we can transform the area integral into sum over paths. The first few terms that give incident radiance at a point $\boldsymbol{p}_0$ from another point $\boldsymbol{p}_1$ where $\boldsymbol{p}_1$ is the first point on a surface along the ray from $\boldsymbol{p}_0$ in direction $\boldsymbol{p}_0 - \boldsymbol{p}_1$:

$$\begin{aligned} L(\boldsymbol{p}_1 \to \boldsymbol{p}_0) = {} & L_e(\boldsymbol{p}_1 \to \boldsymbol{p}_0) \\ & + \int_A L_e(\boldsymbol{p}_2 \to \boldsymbol{p}_1) f(\boldsymbol{p}_2 \to \boldsymbol{p}_1 \to \boldsymbol{p}_0) G(\boldsymbol{p}_2 \leftrightarrow \boldsymbol{p}_1) dA(\boldsymbol{p}_2) \\ & + \int_A \int_A L_e(\boldsymbol{p}_3 \to \boldsymbol{p}_2) f(\boldsymbol{p}_3 \to \boldsymbol{p}_2 \to \boldsymbol{p}_1) G(\boldsymbol{p}_3 \leftrightarrow \boldsymbol{p}_2) \\ & \quad \times f(\boldsymbol{p}_2 \to \boldsymbol{p}_1 \to \boldsymbol{p}_0) G(\boldsymbol{p}_2 \leftrightarrow \boldsymbol{p}_1) dA(\boldsymbol{p}_3) dA(\boldsymbol{p}_2) + \ldots \end{aligned} \tag{101}$$

The third term in the above equation is illustrated in the figure below:

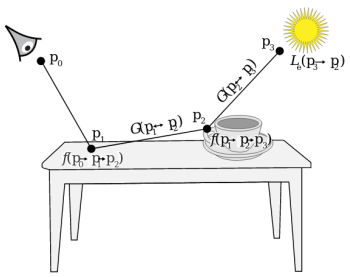


Figure 14: Integral over all points $\boldsymbol{p}_2$ and $\boldsymbol{p}_3$. [32]

Let us discuss the figure above in more detail. We start by sending a ray from our camera denoted by $\boldsymbol{p}_0$ which hits an object in our scene (the desk). The point of the first hit is denoted by $\boldsymbol{p}_1$. Here, we evaluate our BSDF and then send a ray in a random direction in the scene. This ray then eventually hits the cup in our scene at point $\boldsymbol{p}_2$. This hit point at our cup also has its own material properties and so we again evaluate the BSDF of the hit point of the cup and send another ray in random direction. This ray then eventually hits our light source denoted by point $\boldsymbol{p}_3$. These rays that start from the camera and then eventually end at the light source is considered as one single **path** and so we traced a path from our camera to our light source, hence the name **path tracing**.

This path has 4 vertices connected by 3 segments. The total contribution of all such paths of length 4 (i.e., a vertex at the camera, two vertices at points on surfaces in the scene, and a vertex on a light source) is given by this third term. This infinite sum can be written as:

$$L(\boldsymbol{p}_1 \to \boldsymbol{p}_0) = \sum_{n=1}^{\infty} P(\bar{\boldsymbol{p}}_n) \tag{102}$$

$P(\bar{\boldsymbol{p}}_n)$ gives the amount of radiance scattered over a path $\bar{\boldsymbol{p}}_n$ with $n+1$ vertices,

$$\bar{\boldsymbol{p}}_n = \boldsymbol{p}_0, \boldsymbol{p}_1, ..., \boldsymbol{p}_n, \tag{103}$$

where $\boldsymbol{p}_0$ is on the image plane and $\boldsymbol{p}_n$ is on a light source,

$$P(\bar{\boldsymbol{p}}_n) = \underbrace{\int_A \int_A ... \int_A}_{n-1} L_e(\boldsymbol{p}_n \to \boldsymbol{p}_{n-1}) \times (\prod_{i=1}^{n-1} f(\boldsymbol{p}_{i+1} \to \boldsymbol{p}_i \to \boldsymbol{p}_{i-1}) G(\boldsymbol{p}_{i+1} \leftrightarrow \boldsymbol{p}_i) dA(\boldsymbol{p}_2) ... dA(\boldsymbol{p}_n) \tag{104}$$

The product of a path's Bidirectional Scattering Distribution Function (BSDF) f and the geometry terms G above is called the *throughput* of the path; and it is often denoted as:

$$T(\bar{\boldsymbol{p}}_n) = \prod_{i=1}^{n-1} f(\boldsymbol{p}_{i+1} \to \boldsymbol{p}_i \to \boldsymbol{p}_{i-1}) G(\boldsymbol{p}_{i+1} \leftrightarrow \boldsymbol{p}_i) \tag{105}$$

So it can be seen that for each object we encounter at each vertex of the path, we will get the BSDF of the material at that point and multiply it with the geometric term. Combining throughput with equation (101), we have:

$$P(\bar{\boldsymbol{p}}_n) = \underbrace{\int_A \int_A ... \int_A}_{n-1} L_e(\boldsymbol{p}_n \to \boldsymbol{p}_{n-1}) T(\bar{\boldsymbol{p}}_n) dA(\boldsymbol{p}_2) ... dA(\boldsymbol{p}_n) \tag{106}$$

Given (102) and a path length of n, we can compute a monte carlo estimate of the radiance arriving at our camera $\boldsymbol{p}_0$ by sampling a set of vertices with an appropriate sampling density and then evaluate an estimate of $P(\bar{\boldsymbol{p}}_n)$ using those vertices. A better way to create a new path from a previous path is to construct path incrementally. That is, at each vertex, the BSDF is sampled to generate a new direction and the next vertex $\boldsymbol{p}_{i+1}$ is found by tracing a ray from $\boldsymbol{p}_i$ to the sampled direction and find the closest intersection. This is known as **BSDF sampling**. Since we are sampling BSDFs according to solid angle, and path integral LTE is an integral over surface area, we need to convert our pdf from solid angle p_ω to area p_A:

$$p_A(\boldsymbol{p}_i) = p_\omega \frac{|cos\theta_i|}{||p_{i+1} - p_i||^2} \tag{107}$$

Using this sampling technique, we will end our path at the last vertex $\boldsymbol{p}_i$ with a distribution over the surfaces of light sources $p_A(\boldsymbol{p}_i)$, the monte carlo estimate of such path is given by:

$$\frac{L_e(\boldsymbol{p}_i \to \boldsymbol{p}_{i-1}) f(\boldsymbol{p}_i \to \boldsymbol{p}_{i-1} \to \boldsymbol{p}_{i-2}) G(\boldsymbol{p}_i \leftrightarrow \boldsymbol{p}_{i-1})}{p_A(\boldsymbol{p_i})} \times (\prod_{j=1}^{i-2} \frac{f(\boldsymbol{p}_{j+1} \to \boldsymbol{p}_j \to \boldsymbol{p}_{j-1}) |cos\theta_j|}{p_\omega(\boldsymbol{p}_{j+1} - \boldsymbol{p}_j)}) \tag{108}$$

5.6.3 Russian Roulette

If you look closely at the path integral formulation (106) we arrived in the previous section, it can be seen that it is an integral over paths. But an important question that needs to be answered is how many paths or bounces are enough? It is also important that if we want to be physically correct, we can't explicitly stop bouncing once we reach a specific path length. An unbiased physically based renderer would only stop the bouncing of light when the light itself is entirely absorbed but the problem is that no material in the world actually absorbs 100% of the light. This means that no matter how long our path extends, the probability that the light is entirely absorbed would never be zero.

It has been observed though that most of the contribution of the overall radiance computed at a path comes from the first few bounces. Intuitively, this can be thought of this way: if an object is close to the light source, that specific object would receive the most amount of light and an object that is not exactly near the light source but at a distance from the object that receives the light, would receive some light from the light source and some amount of light reflected off from the first object. The first object would also receive the reflected light from the second object but the intensity would be significantly less. So in other words, as light bounces through the scene, it gets dimmer.

Russian roulette exploits exactly this fact we discussed above. We first decide on a probability p_{RR} to continue our path. Then we draw a random value x in $[0, 1)$ and decide on this value if we should keep going for another bounce or end our path:

$$x < p_{RR} : \quad keep \;\; going \;\; for \;\; another \;\; bounce \tag{109}$$

$$x \geq p_{RR} : \quad end \;\; path \tag{110}$$

The longer a path goes, the more likely it is to get terminated. Some paths if lucky, can still survive for an extended length since the method is unbiased and only limited on the floating point precision of a machine. It is also interesting to note that as the path gets longer, the contribution to the final radiance estimate along that path is also higher. One heuristic for path termination probability is to actually use the maximum RGB coefficient of the throughput T (105) of the path. Remember that throughput was the product of all the BSDFs encountered in our path and the geometric term. This corresponds the probability p_{RR} with the intensity of the throughput value in our path which makes sense because if a path is carrying a throughput value which is close to zero; it basically means that the amount of impact it would have in our final radiance estimate would also be zero so paths that contribute very little to the pixel in our image are likely to terminate earlier. We compute p_{RR} as follows:

$$p_{RR} = min(max(T(\bar{\boldsymbol{p}}_n)), 0.99) \tag{111}$$

It is recommended to start using russian roulette after atleast three bounces in order to avoid terminating very short paths which would lead to a lot of variance usually. We will discuss this more in the analysis section. There is still one last problem to fix when using (111). Consider tracing a path along very bright mirror surfaces that absorb less light; for a case like this, our throughput stays high for an extended period of time. One good way to handle such case is to adjust our throughput by dividing it by the termination probability p_{RR}. Notice that we also clamp p_{RR} with a value 0.99 just to limit bouncing forever. The psuedocode for brute force path tracing using russian roulette is as follows:

Require: scene S, sampler s, ray r
Ensure: color pct

function BRUTEFORCEINTEGRATOR(S, s, r)
    $pct \leftarrow$ Color(0, 0, 0) ▷ pct is total path contribution
    $T \leftarrow$ Color(0, 0, 0) ▷ T is throughput
    $pL \leftarrow 0$ ▷ pL is path length
    $r_c \leftarrow r$ ▷ r_c is the ray that holds different paths
    while S.intersect(intersection, r_c) **do**
        if intersection.mesh is emitter **then**
            $pct \leftarrow pct+$ intersection.mesh.emitter.radiance() $\times T$
                ▷ create a BSDF record $fRec$ by passing it the the opposite ray direction $-r_c.dir()$
        $T \leftarrow T\times$ intersection.mesh.BSDF.sample($fRec$, $s.2D()$)
        if $pL > 3$ **then**
            $p_{RR} \leftarrow$ min(T.maxCoefficient(), 0.99)
            $T \leftarrow \frac{T}{p_{RR}}$
            if $s.1D() > p_{RR}$ **then**
                break ▷ break out of the loop and stop bouncing
        $r_c \leftarrow$ Ray(intersection.position, $fRec.\omega_o$) ▷ create a new ray from intersection position in the outgoing direction of BSDF
        $pL \leftarrow pL + 1$
    return pct

5.6.4 Next Event Estimation

5.6.4.1 Concept

It is obvious to see why a brute force path tracer would have a lot of variance in our images. We are only sampling the BSDF in the approximation of our integrator. We also stop bouncing of rays using russian roulette, this would mean that for many of the paths, we might not even hit the light source. For cases like these, the computed pixel shows up as noise in our images. The next thing we have to improve our results is to increase the number of samples. This does reduce variance but as we take more and more samples, it becomes harder and harder to reduce noise further. We will see this in detail in the analysis section. The question for now is, if we can do better than just sampling the BSDF? The idea that could help us here comes from what we already implemented when we did Whitted Style Ray tracing. When we computed direct illumination, we explicitly sampled our light source to compute the amount of light reaching a point in our scene. What if we can combine direct illumination with what we have in our brute force path tracer? only this time, we will be doing it recursively for each hit point in our scene along with BSDF sampling. So for each hit in our scene:

1. We sample the BSDF.
2. We also explicitly send a shadow ray towards the light source and compute the amount of direct illumination reaching at that bounce. This is called *light sampling* or *emitter sampling*.

Intuitively this makes sense, because when we're doing BSDF sampling, we're asking ourselves, if we assume that the light is uniform, which directions contribute the most for our monte carlo estimate? Remember the ambient occlusion algorithm which assumed that a diffuse surface receives illumination from all directions. In the same way, when we do light sampling, we're asking ourselves, if we now know where the light is, which directions might actually hit them and contribute to our estimate? Previously, the amount of light computed at each bounce relied on coincidence but this time, we're actually computing that light explicitly. In a way, we have separated indirect illumination (BSDF sampling) and direct illumination (light sampling) and using the fact that indirect light is basically direct light from the few bounces we'll compute in the future. This is known as **Next event estimation**.

5.6.4.2 Implementation

So in order to start with next event estimation we have to split our integral into two parts. It is splitted into an integral to compute 1) direct illumination and 2) indirect illumination. For direct illumination, we will follow the same process we had in direct illumination section, i.e we intersect with a point in our scene and compute radiance by generating a shadow ray from that point towards the emitter. After that, for the indirect illumination, we create an extra ray that is a result of BSDF sampling. So depending on the type of BSDF the ray hits, we sample another ray which eventually hits or does not hit another point in the scene. This is the same as we did in our brute force path tracer. This keeps on happening until the path is terminated using russian roulette.

There is one case which needs to be handled explicitly. Since we are generating shadow rays and computing direct illumination explicitly at each path, there would be some cases where our indirect ray might also hit an emissive surface which will give the same radiance that we have already computed earlier using direct illumination. This would result in *double counting* as shown below:

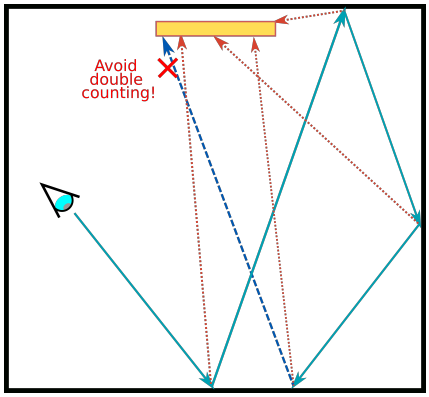


Figure 15: A path that computes direct and indirect illumination separately.
An indirect path might hit an emissive surface resulting in *double counting*.

In the figure above, the blue arrows indicate the indirect bounces in our scene, the red arrows indicate the explicit light sampling at each bounce. After a couple of bounces, one of the indirect bounces also hit the light source which eventually as the path ends would result in twice the amount of light we wanted to compute violating the law of energy conservation.

To tackle this, we need to check if the intersection we had indirectly is actually an emitter. If we do have an intersection with an emitter then we only get the radiance from that and skip the explicit computation of direct illumination. Traditionally what is done is that we only do light sampling for diffuse BSDFs and skip it for specular BSDFs. When we do hit a specular surface indirectly, we consider its emissive color in the next bounce. The reason to do this is because specular BSDFs are delta BSDFs;

meaning that they only reflect light in specific directions while being zero in all other directions. The probability of such a BSDF to have any contribution when directly sampling it with light source will also always be zero. Now we are ready to write the monte carlo approximation for implementing next event estimation. In very simple form the equation is:

$$L_r(\boldsymbol{p}) = L_e + L_{direct} + L_{indirect} \tag{112}$$

So, both L_{direct} and $L_{indirect}$ can be written as separate integrators:

$$L_r(\boldsymbol{p}) = L_e + \int_A ...L_e(\boldsymbol{p} \leftarrow \boldsymbol{p'})...dA_e(\boldsymbol{p'}) + \int_\Omega ...L(\boldsymbol{p}, \boldsymbol{\omega'})...d\boldsymbol{\omega'} \tag{113}$$

And similarly we can write these integrators as monte carlo estimators as follows:

$$\begin{aligned} L_r(\boldsymbol{p}) = L_e + \\ &\frac{1}{N}\sum_{k=1}^{N}\frac{f_r(\boldsymbol{p}_k, \boldsymbol{p}_k \rightarrow \boldsymbol{p'}, \boldsymbol{\omega}_r)G(\boldsymbol{p}_k \leftrightarrow \boldsymbol{p'})L_e(\boldsymbol{p'}, \boldsymbol{p'} \rightarrow \boldsymbol{p}_k)}{p_{\mathcal{L}}(\boldsymbol{p}_k)} \\ &+ \frac{1}{N}\sum_{k=1}^{N}\frac{f_r(\boldsymbol{p}_k, \boldsymbol{p}_k \rightarrow \boldsymbol{p'}, \boldsymbol{\omega}_r)L_i(\boldsymbol{p}_k, \boldsymbol{p}_k \rightarrow \boldsymbol{p'})cos\theta_k d\boldsymbol{\omega}_k}{p_\Omega(\boldsymbol{p}_k)} \end{aligned} \tag{114}$$

And the pseudocode for next event estimation is as follows:

Require: scene S, sampler s, ray r
Ensure: color pct

function NEEIntegrator(S, s, r)
  $pct \leftarrow$ Color$(0, 0, 0)$ ▷ pct is total path contribution
  $T \leftarrow$ Color$(0, 0, 0)$ ▷ T is throughput
  $pL \leftarrow 0$ ▷ pL is path length
  $r_c \leftarrow r$ ▷ r_c is the ray that holds different paths
  $doEmit \leftarrow true$ ▷ $doEmit$ is a boolean that considers emission if required
  while S.intersect(intersection, r_c) **do**
    if intersection.mesh is emitter and $doEmit$ is true **then** ▷ consider emission
      $pct \leftarrow pct + L_i$+ intersection.mesh.emitter.radiance() $\times T$
    if intersection.mesh.BSDF is diffuse **then**
      $light_{pdf} \leftarrow 0$ ▷ sample an emitter from the scene and save it as em
      $light_p \leftarrow$ em.sampleUniform($light_{pdf}$)
      $light_{\omega_o} \leftarrow (light_p -$ intersection.position()).normalized() ▷ get normalized outgoing light direction in $light_{\omega_o}$
      $L_i \leftarrow \frac{incidentLightRadiance(em)}{light_{pdf}}$ ▷ compute light radiance as explained in area lights section
      ▷ create a BSDF record $lightFRec$ by passing it the incoming ray direction $-r_c.dir()$ and outgoing direction $light_{\omega_o}$
      $pct \leftarrow pct + L_i \times$ intersection.mesh.BSDF.eval($lightFRec$) $\times T$
      $doEmit \leftarrow false$
    else
      $doEmit \leftarrow true$
      ▷ create a BSDF record $fRec$ by passing it the incoming ray direction $-r_c.dir()$
    $T \leftarrow T \times$ intersection.mesh.BSDF.sample($fRec$, $s.2D()$)
    if $pL > 3$ **then**
      $p_{RR} \leftarrow$ min(T.maxCoefficient(), 0.99)
      $T \leftarrow \frac{T}{p_{RR}}$

24: **if** $s.1D() > P_{RR}$ **then**
25: **break** ▷ break out of the loop and stop bouncing
26: $r_c \leftarrow$ Ray(intersection.position, $fRec.\omega_o$) ▷ create a new ray from intersection position in the outgoing direction of BSDF
27: $pL \leftarrow pL + 1$
28: **return** pct

5.6.5 Multiple Importance Sampling

5.6.5.1 Concept

If we look back at our rendering equation (13). It can be seen that it is a product of different functions. At first glance we see the BSDF f_r and the incoming light L_i. Let's just consider an integrand with a product of two functions $f(x)$ and $g(x)$: $\int f(x)g(x)$. If we are able to importance sample both $f(x)$ and $g(x)$, what is the best way to combine both the sampling strategies to compute a PDF that is proportional to the product $f(x)g(x)$? Remember the monte carlo estimator we discussed previously (32). We discussed that placing samples intelligently at locations where the integrand is high would help us bring closer to the integrand we are approximating. But when we have different importance sampling strategies, there might be a case that carefully placing samples for one strategy might not work for another hence variance would be high. Generally, if the function value at the sample we chose is high and the pdf at that same sample has a low value, this is one of the worst cases that we would like to avoid when using multiple strategies and it can happen quite often as we will discuss in the analysis section. The idea behind MIS is to combine multiple sampling strategies in a way that it results in lower variance. This is true because if our samples are carefully placed where they are supposed to be, we should be able to come closer to the original integrand we want to approximate. If one strategy covers one major part of the integrand and the other covers another major part of the integrand, combining them intelligently should reduce the overall variance.

This however needs to be done carefully as we want to sample both the strategies but also do it in a way so as to reduce the overall variance. This is not an easy task because when sampling two strategies, if either of them is sampled naively, it can result in high variance (noisy) image. This is because variance is additive [27]. So if we have two distributions, p_1 and p_2, according to which we need to approximate our estimator, given the sample $\boldsymbol{x}_i$, a naive combination of 2 sampling strategies would look like:

$$<F^{N^1+N^2}> = \frac{w_1}{N_1}\sum_{i=1}^{N}\frac{f(\boldsymbol{x}_i)}{p_1(\boldsymbol{x}_i)} + \frac{w_2}{N_2}\sum_{i=1}^{N}\frac{f(\boldsymbol{x}_i)}{p_2(\boldsymbol{x}_i)} \tag{115}$$

So we can see that both the strategies are weighted using w_1 and w_2. But we don't want to weight the estimators, and instead want to weight the individual samples that come from both the estimators. So these weights can also take the sample and evaluate a value for each of the strategies; so rewriting equation (115):

$$<F^{N^1+N^2}> = \frac{1}{N_1}\sum_{i=1}^{N}w_1(\boldsymbol{x}_i)\frac{f(\boldsymbol{x}_i)}{p_1(\boldsymbol{x}_i)} + \frac{1}{N_2}\sum_{i=1}^{N}w_2(\boldsymbol{x}_i)\frac{f(\boldsymbol{x}_i)}{p_2(\boldsymbol{x}_i)} \tag{116}$$

$$where \quad f(\boldsymbol{x}_i) \neq 0 \implies w_1(\boldsymbol{x}) + w_2(\boldsymbol{x}) = 1 \tag{117}$$

$$and \ \ if \ \ p_1(\boldsymbol{x}_i) = 0, \ \ p_2(\boldsymbol{x}_i) = 0 \implies w_1(\boldsymbol{x}) = 0, \ \ w_2(\boldsymbol{x}) = 0 \tag{118}$$

We can now generalise this to M strategies instead of just 2 strategies:

$$<F^{\sum N_s}> = \sum_{s=1}^{M} \frac{1}{N_s} \sum_{i=1}^{N_s} w_s(\boldsymbol{x}_i) \frac{f(\boldsymbol{x}_i)}{p_s(\boldsymbol{x}_i)} \tag{119}$$

where again:

$$f(\boldsymbol{x}) \neq 0 \implies \sum_{s=1}^{M} w_s(\boldsymbol{x}) = 1 \tag{120}$$

$$and \ \ if \ \ p_s(\boldsymbol{x}) = 0 \implies w_s(\boldsymbol{x}) = 0 \tag{121}$$

Now how do we actually choose these weights? There are many proposed methods for this but there are some heuristics that can be used to calculate "good" values for our weights. One of the heuristic is known as the **balance heuristic** [39]:

$$w_s(\boldsymbol{x}) = \frac{N_s p_s(\boldsymbol{x})}{\sum_j N_j p_j(\boldsymbol{x})} \tag{122}$$

So the weight w with strategy s at the part of the domain $\boldsymbol{x}$ is equal to the number of samples with our strategy N_s multiplied by the pdf value of generating that location $p_s(\boldsymbol{x})$ divided by the weighted sum of all pdfs from all strategies we are using multiplied by the number of samples we allocate for those strategies. Another commonly used heuristic is the **power heuristic**. It calculates weights as:

$$w_s(\boldsymbol{x}) = \frac{[N_s p_s(\boldsymbol{x})]^\beta}{\sum_j [N_j p_j(\boldsymbol{x})]^\beta} \tag{123}$$

Intuitively, power heuristic works similarly to the balance heuristic but sharpens the weights making small contributions smaller and large contributions larger. For instance, for a case when one of the strategies is already a very close approximate of the integrand and the other strategy only helps us a little bit, balance heuristic might make things worse and power heuristic might be the optimal choice. You may notice that when $\beta = 1$ in equation (123), then we indeed have the balance heuristic.

5.6.5.2 Implementation

In our case, we will be combining the two strategies we have so far i.e BSDF sampling and light sampling. When generating a sample with either of the two strategies, we will weight them with their PDF. For weighting the contribution of the light source sample, the weighting function looks like this:

$$w_{\mathcal{L}}(p_{\mathcal{L}}, p_\Omega) = \frac{p_{\mathcal{L}}}{p_{\mathcal{L}} + p_\Omega} \tag{124}$$

It is important to note that the above weighting function only works if both the probabilities are expressed in the same unit (solid angle or unit area). For example, for a diffuse BSDF the pdf according to solid angle is given by (74) and for the light source this would be converted as:

$$p_{\mathcal{L}} = \frac{1}{Area_{source}} \frac{r^2}{\boldsymbol{n} \cdot \boldsymbol{\omega_o}} \tag{125}$$

where $Area_{source}$ is the surface area of the light source, $\boldsymbol{n}$ is the normal of the point being shaded, $\boldsymbol{\omega_o}$ is the outgoing light direction and r^2 is the squared distance from the point being shaded and the light source. Now when we have to weight the BSDF sampling, we check if the outgoing direction from the BSDF sample hits the light source by sending a shadow ray and if it does hit a light source, we weight the contribution as:

$$w_\Omega(p_{\mathcal{L}}, p_\Omega) = \frac{p_\Omega}{p_{\mathcal{L}} + p_\Omega} \tag{126}$$

Given these weights, we use the estimator (114) but this time adding our weights:

$$L_r(\boldsymbol{p}) = L_e + \frac{1}{N}\sum_{k=1}^{N} w_{\mathcal{L}(\boldsymbol{p}_k)} \frac{f_r(\boldsymbol{p}_k, \boldsymbol{p}_k \to \boldsymbol{p'}, \boldsymbol{\omega}_r) G(\boldsymbol{p}_k \leftrightarrow \boldsymbol{p'}) L_e(\boldsymbol{p'}, \boldsymbol{p'} \to \boldsymbol{p}_k)}{p_{\mathcal{L}}(\boldsymbol{p}_k)} + \frac{1}{N}\sum_{k=1}^{N} w_{\Omega(\boldsymbol{p}_k)} \frac{f_r(\boldsymbol{p}_k, \boldsymbol{p}_k \to \boldsymbol{p'}, \boldsymbol{\omega}_r) L_i(\boldsymbol{p}_k, \boldsymbol{p}_k \to \boldsymbol{p'}) cos\theta_k d\boldsymbol{\omega}_k}{p_{\Omega}(\boldsymbol{p}_k)} \tag{127}$$

And the pseudocode for the multiple importance sampling estimator is as follows:

Require: scene S, sampler s, ray r
Ensure: color pct

function MISIntegrator(S, s, r)
  $pct \leftarrow$ Color$(0, 0, 0)$ ▷ pct is total path contribution
  $T \leftarrow$ Color$(0, 0, 0)$ ▷ T is throughput
  $pL \leftarrow 0$ ▷ pL is path length
  $r_c \leftarrow r$ ▷ r_c is the ray that holds different paths
  $w_\Omega \leftarrow 1$ ▷ weight for BSDF sampling
  while S.intersect(intersection, r_c) **do**
    if intersection.mesh is emitter **then** ▷ consider emission
      $pct \leftarrow pct + L_i +$ intersection.mesh.emitter.radiance() $\times T \times w_\Omega$ ▷ do light importance sampling
    if intersection.mesh.BSDF is diffuse **then**
      $light_{pdf} \leftarrow 0$ ▷ sample an emitter from the scene and save it as em
      $light_p \leftarrow$ em.sampleUniform($light_{pdf}$)
      $light_{\omega_o} \leftarrow (light_p -$ intersection.position()).normalized() ▷ get normalized outgoing light direction in $light_{\omega_o}$
      $L_i \leftarrow \frac{incidentLightRadiance(em)}{light_{pdf}}$ ▷ compute light radiance as explained in area lights section
      ▷ create a BSDF record $lightFRec$ by passing it the incoming ray direction $-r_c.dir()$ and outgoing direction $light_{\omega_o}$
      $bsdf_{pdf} \leftarrow intersection.mesh.BSDF.pdf(lightFRec)$ ▷ get BSDF pdf
      $w_{\mathcal{L}} \leftarrow \frac{light_{pdf}}{light_{pdf} + bsdf_{pdf}}$
      $pct \leftarrow pct + L_i \times$ intersection.mesh.BSDF.eval($lightFRec$) $\times T \times w_{\mathcal{L}}$
      ▷ create a BSDF record $fRec$ by passing it the incoming ray direction $-r_c.dir()$
    $T \leftarrow T \times$ intersection.mesh.BSDF.sample($fRec$, $s.2D()$) ▷ create a new ray from intersection position in the outgoing direction of BSDF
    $r_c \leftarrow$ Ray(intersection.position, $fRec.\omega_o$)
    $shadowRay_{hit} \leftarrow S.intersect(intersection2, r_c)$ ▷ do BSDF importance sampling
    if $shadowRay_{hit}$ is true and intersection2.mesh is emitter **then** ▷ create an emitter record $em2$ by passing it intersection2 and intersection

      $light_{pdf2} \leftarrow getLightPdf(intersection.mesh, em2)$
      $L_i \leftarrow incidentLightRadiance(em2)$ ▷ compute light radiance as explained in area lights section

      $bsdf_{pdf2} \leftarrow intersection2.mesh.BSDF.pdf(fRec)$ ▷ get BSDF pdf
      if intersection2.mesh.BSDF is diffuse **then**

29: $w_\Omega = \frac{bsdf_{pdf2}}{light_{pdf2}+bsdf_{pdf2}}$

30: $pct \leftarrow pct + L_i \times w_{Omega}$

31: **if** $pL > 3$ **then**

32: $p_{RR} \leftarrow \min(T.\text{maxCoefficient}(), 0.99)$

33: $T \leftarrow \frac{T}{p_{RR}}$

34: **if** $s.1D() > P_{RR}$ **then**

35: **break** ▷ break out of the loop and stop bouncing

36: $pL \leftarrow pL + 1$

37: **return** pct

5.7 Tone Mapping

The range of light that we see in the real world spans ten orders of absolute range. This is to say that the *dynamic range* of the light in the real world is considerably larger than what we can reproduce in normal image formats. *Dynamic range* in photography for example, is the ratio between the maximum and minimum measurable light intensities [13]. It might be surprising to know that we never really encounter true white or true black in the real world, and only see varying degrees of light reflectivity. In path tracing, where we are computing radiance in our virtual scenes, the image constructed would be too difficult to reproduce on a modern day monitor which only supports a fraction of dynamic range of what our human eyes are capable of. For instance most modern day monitors support RGB values in the range of [0, 255], but most renderers (including our path tracer) output radiance values in [0, ∞) [1]. We call such values high dynamic range (HDR) while the viewable target range is low dynamic range (LDR). It is important that the dynamic range is compressed so that the brightest areas do not look too bright and the darkest areas do not look too dark. Moreover, as we've already discussed this before, since we are approximating the rendering equation, there might be cases where some scenes would end up with noise in our images, even with a large number of samples. **Tone mapping** helps us in this case which is a post processing step we apply before saving our final rendered image. It also helps us to make our images more appealing since it adjusts the brightness and contrast of our image in such a way that more areas of our rendered image are visible when seen on our monitor. The renderer we are using, Nori, converts the final radiance estimate we get for all our pixels to standard RGB (sRGB) which is supported by most modern day monitors. But we will talk about one another technique that is extensively being used in computer graphics, known as the **Extended Reinhard Luminance Tone Mapper**[34] [30].

5.7.1 Extended Reinhard Luminance Tone Mapper

A tone mapper essentially is a function that takes in RGB values and outputs another set of these values which are more suitably converted to support our monitor's dynamic range and to show as much detail in our scene as possible. This needs to be done carefully though since just clamping the values would result in loss of detail. The extended reinhard luminance tone mapper works on *luminance* instead of RGB triplets. Luminance is the amount of light that passes through, is emitted from, or is reflected from a particular area, and falls within a given solid angle [45]. In simpler terms, it is a scalar value that tells us how bright we view something. We also work on sRGB instead of RGB in this tone mapper. Converting an sRGB value to luminance is as follows:

$$L = 0.2126R + 0.7152G + 0.0722B \tag{128}$$

Extended Reinhard converts our sRGB radiance to luminance, applies tone mapping on the luminance value and then scale our RGB value according to the new luminance value. The scaling is done as:

$$C_{out} = C_{in}\frac{L_{out}}{L_{in}} \tag{129}$$

where C_{in} is the old RGB value, L_{in} is the converted luminance of C_{in} using equation (128) and L_{out} is the maximum luminance value in our image. To calculate L_{out} we check for each pixel of our image what luminance we get using equation (128) and find the maximum among all pixels. Next, we compute the numerator value for computing L_{new} (the new luminance value) as:

$$n = L_{in} * (1 + \frac{L_{in}}{L_{out}^2}) \tag{130}$$

And L_{new} is computed as:

$$L_{new} = \frac{n}{1 + L_{in}} \tag{131}$$

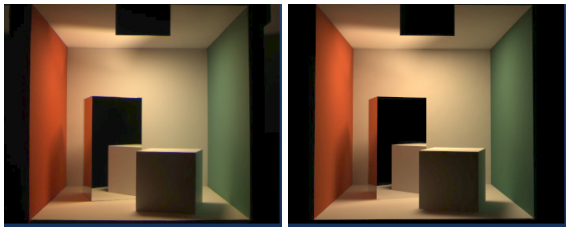
And changing back to sRGB is done by using equation (129) but this time passing L_{new} as L_{in}. We then use C_{out} as the new RGB value for our pixel.

6 Analysis

Analysis of a physically based renderer plays an important role in getting useful insights about how capable the renderer is. This is because there are infinite possibilities of creating scenes where the path tracer could perform really well, creating highly realistic images with little noise but there are equal number of scenarios where the same path tracer could fail to give reasonable results. It is therefore useful to analyze a renderer in different scenarios which can then help us gain knowledge on how to improve on an existing approach. Since we are approximating the rendering equation by monte carlo integration using multiple importance sampling, we are aware that taking different number of samples would result in images with increasing quality (24) i.e the higher the number of samples, the less noisy the image looks. It is important to note again that different scenes would look differently given a specific number of samples. This means that if one scene with different lighting conditions and different types of objects was rendered using 128 samples per pixels and another scene with different conditions was also rendered using the same spp, it is possible that the first scene could be less noisy than the later or vice versa. Let us now review some ways to quantify error in a light transport algorithm.

6.1 Calculating error in rendering

Since, we are using a monte carlo estimator to estimate the reflectance integral in our rendering equation, which as the name suggests *"estimates"* the real value of the integral, there will always be error in our rendered images. In fact, it is well known that monte carlo methods has a variance of $\Theta(1/N)$ where N is the number of samples [19] [32]. For a rendered image, the resultant color of the pixel obtained can be used to calculate the error using the same pixel in a reference image of the same scene. A reference image (often known as ground truth), would be a photograph taken in the real world through a camera. For example, the original Cornell box was created physically by taking a photograph under controlled lighting conditions using a Photometrics CCD camera. And then the same image was rendered using a geometric model with material properties and lighting set to values identical to the physical conditions.



(a) Real photograph (b) Simulated

Figure 16: The original Cornell box along with the simulated rendered image. [16].

But it might be obvious that getting such images which can also be replicated in virtual rendered scenes is often very difficult, especially when we're able to render highly complex scenes nowadays. So in most cases, another high quality rendered image is then used as a ground truth. For instance we can use a rendered image with the same path tracer we're using but with a very high samples per pixels (spp) count. We then calculate the error using our estimator $\hat{I}$ and the ground truth I as:

$$\epsilon = ||\hat{I} - I|| \tag{132}$$

The above is called *absolute error* but a better and more traditional method of measuring error in renderings is calculating the *Mean Squared Error (MSE)*. To have a better intuition about MSE, let us first define *Bias*.

6.1.1 Bias

Bias of a monte carlo estimator is the expected value of its error. Or, it is the difference between the expected value and the true value I:

$$Bias[\hat{I}] = E[\hat{I} - I] = E[\hat{I}] - I \tag{133}$$

You can observe from the above equation that an estimator whose expected value equals I has zero bias and is thus called *unbiased*. This is same as saying that if an estimator on average, gives a result that is directly identical to our reference value, then the renderer using that estimator can be called as a perfectly unbiased renderer.

6.1.2 Mean Sqaured Error (MSE)

The mean squared error (MSE) of an estimator calculates its average squared error:

$$MSE[\hat{I}] = E[(\hat{I} - I)^2] \tag{134}$$

Or, expanding the above equation and writing it in a different form, MSE is the variance of the estimator plus the square of its bias:

$$MSE[\hat{I}] = Var[\hat{I}] + Bias[\hat{I}]^2 \tag{135}$$

In order to compute MSE, we often take multiple images rendered by the same renderer with same samples per pixel and then compute the total MSE using (134).

6.1.3 Root Mean Squared Error (RMSE)

Root Mean Squared Error (RMSE) is also often used since it is less sensitive to outliers in pixels of our image. Outliers in a rendered image could be dead pixels where the radiance estimate was too high. We calculate RMSE by taking the square root of MSE:

$$RMSE[\hat{I}] = \sqrt{MSE[\hat{I}]} \tag{136}$$

6.1.4 Relative Error (RE)

Another error known as *relative error* is also often used in rendering. This is computed as:

$$RE[\hat{I}] = \frac{\hat{I} - I}{I} \tag{137}$$

Note that we are not squaring or taking the absolute value when computing this error. This means that we can often end up with values that are negative. Once we have relative error, we can compute *Relative Mean Squared Error (RMSE)*, *Absolute Error* or *Relative Root Mean Squared Error (RRMSE)*.

6.1.5 Proxy Algorithm

A *proxy algorithm* was discussed in papers **Quantifying the convergence of light transport algorithms** [10] **Quantifying the error of light transport algorithms** [11]. The paper proposes to configure an unbiased renderer to produce N short renderings $\hat{I}_i$ and compute the final solution $\hat{I}_N$ as an average:

$$\hat{I}_N = \frac{1}{N} \sum_{i=1}^{N} \hat{I}_i \tag{138}$$

We then use the result obtained above to compute our errors. The short renderings we use should also be done on multiple levels, so we generate N short renderings with a sample count of 4, and get the averaged result. Then we do another round with a sample count of 8, and keep going to get multiple averaged results using increasing amount of sample count. It has been observed that the error computed using the averaged result is comparable to measuring error of a rendering with large rendering times (with a large sample count). It was also observed in the paper by looking at individual pixels of the short renderings, that the preconditions of the Central Limit Theorem (CLT) were satisfied:

1. The short renderings are independent.
2. Corresponding pixels share common probability distributions, because they were generated by the same scene, camera and algorithm setup, and only using different random numbers.
3. Their expectations are defined.
4. Their variance is finite.

6.1.6 Visualizing error

A paper by NVIDIA **ꟻLIP: A Difference Evaluator for Alternating Images** [2] proposes a difference evaluator with a particular focus on the differences between rendered images and corresponding ground truths. It has a focus on producing an image that approximates the difference perceived by humans when alternating between two images. The evaluator is also able to pay attention to differences in point-like structures, such as **fireflies**, in rendered images i.e., isolated pixels with colors that differ

greatly from the color of their surroundings. The output that is generated is a new image indicating the magnitude of the perceived difference between two images at each pixel. The pixel values are approximately proportional to the perceived error in the test image. We refer the reader to the original paper [2] in order to read more about how the evaluator works. For an overview, here is how the pipeline looks like for the algorithm.

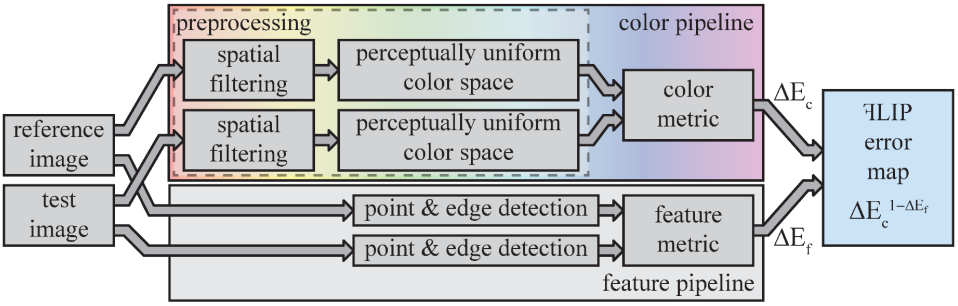
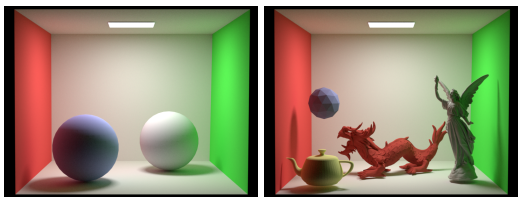


Figure 17: The algorithm pipeline for ꟻLIP [2].

6.2 Test scenes

We now discuss some of the test scenes we will be using to do our analysis. Arguably the most famous scene that is being used throughout the history of computer graphics to evaluate rendering algorithms is the Cornell Box [22]. It is a cube with diffuse walls with an area light at the top. The scene could be tested with different models placed inside the cube and see how our path tracer performs. One important quality of the Cornell Box is the fact that we can observe effects such as color bleeding, reflections, refractions very easily as all objects are confined in a box. For one of our scenes, we will start with 2 diffuse spheres inside the cube. For another scene, we'll test the famous stanford asian dragon [37], stanford lucy [36], utah teapot [31] and an icosphere which was generated in blender [6]. All scenes will be tested starting from 4 samples per pixels to 512 samples per pixels using multiple importance sampling with balance heuristic and next event estimation. All results were obtained on a pc with a Quad Core AMD Ryzen 5 PRO 3500U CPU and 16GB of RAM. The images were rendered using Nori [25] by implementing all the integrators as well as other functionalities mentioned previously. All objects in these two scenes are using a diffuse BRDF.

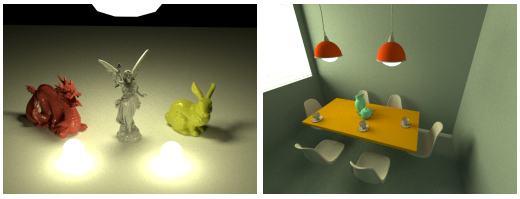


(a) 512spp with MIS (b) 512spp with MIS

Figure 18: Left) Spheres in Cornell box. Right) Utah teapot, Stanford Asian Dragon, Stanford Lucy and an icosphere in Cornell box.

We would also like to evaluate how the path tracer performs in a scene with an open environment. But also a scene that closely resembles a room with objects in it. For this purpose, we created 2 more scenes. One of the scene contains stanford asian dragon, stanford lucy and stanford bunny [35] on a plane but in an open environment. 3 area light sources illuminate the scene. The 3 objects are modeled using a microfacet model with interior index of refraction η_i value of 1.7 and α value of 0.08. They closely resemble a polished stone like material. The other scene is a slightly changed variant of the

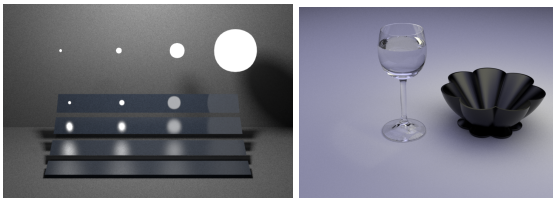
breakfast room [5] where light comes into the room through a window. The light is modeled as an area light outside the window. Two area lights also illuminate the room.



(a) 512spp with MIS (b) 512spp with MIS

Figure 19: Left) Asian Dragon, Lucy, Bunny on a plane. Right) The breakfast room [5].

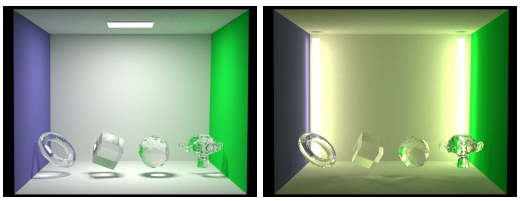
Next we have the scene from Eric Veach [39]. This scene is particularly used to test robustness of multiple importance sampling i.e if we're able to efficiently use bsdf sampling as well as light sampling. The scene has 4 panels modeled using a microfacet material. The panels become shinier or glossier from bottom to top. There are 4 area lights each with varying size from left to right. One another scene contains 2 objects on a plane. One object is a dielectric glass with a water medium in it. Remember that water is also a dielectric material. We just need to model it with different reflective and refractive indices. For the water-air interface, η_i is 1.33 and η_e is 1; for the glass-air interface, η_i is 1.5 and η_e is 1 and for the glass water interface, η_i is 1.33 and η_e is 1.5. The other object is a bowl modeled as a microfacet model with a k_d value of $(0, 0, 0)$. Both of these scene were provided by Nori [25] for testing purposes.



(a) 512spp with MIS (b) 512spp with MIS

Figure 20: Left) Scene from Eric Veach [39]. Right) A glass full of water and a bowl on a plane [25].

Next we go back to the Cornell Box again but this time have different objects as dielectric. Here we will test 2 scenarios by changing the lighting conditions. The first variant is with the traditional area light at the top. The other variant uses two cylinder area lights at the two corners of the box. This helps us analyze the same scene and see how the noise differs by changing the lighting conditions.

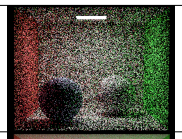
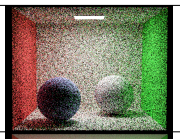
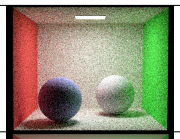
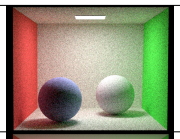
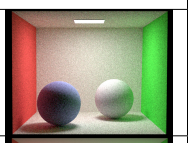
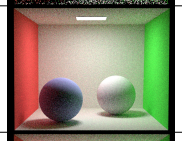
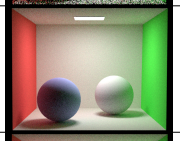
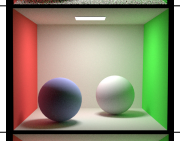
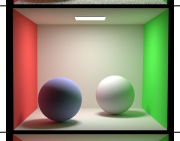
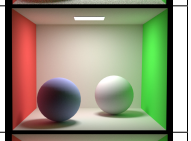
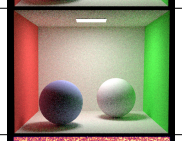
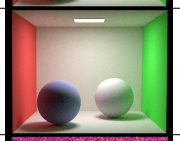
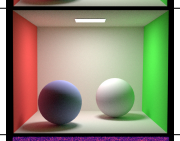
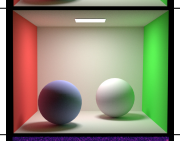
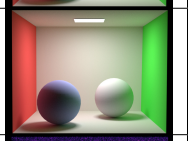
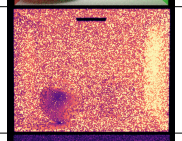
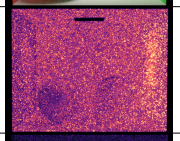
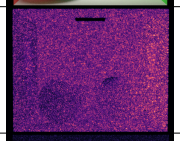
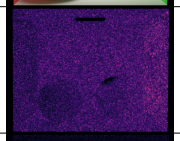
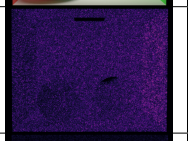
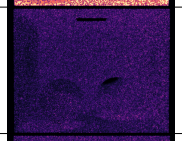
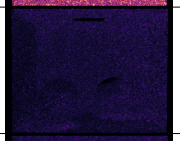
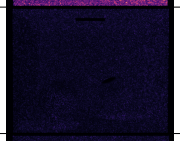
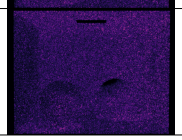
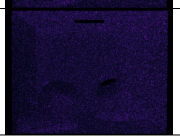
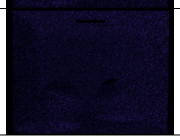


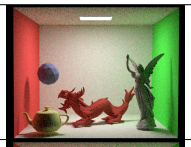
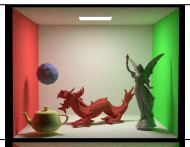
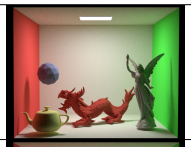
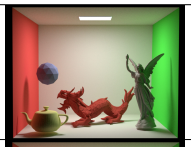
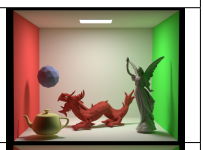
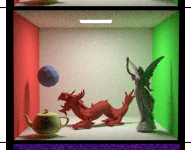
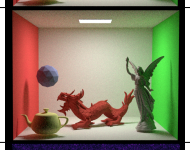
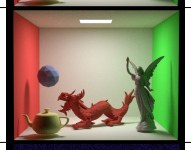
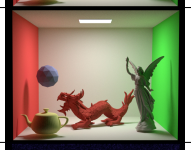
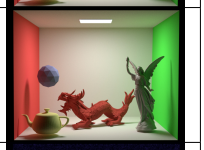
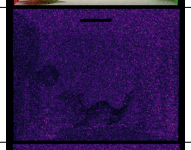
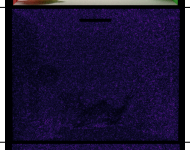
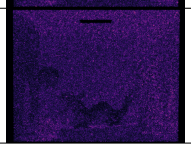
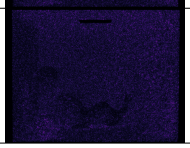
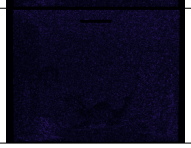
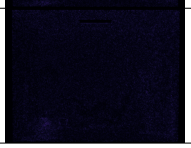
(a) 512spp with MIS (b) 512spp with MIS

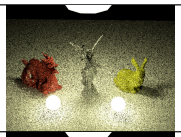
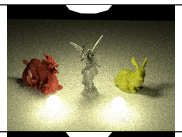
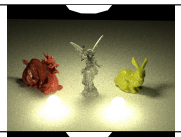
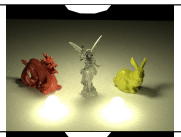
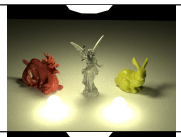
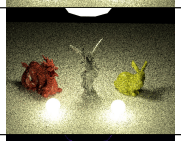
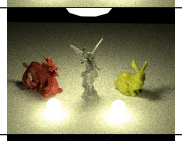
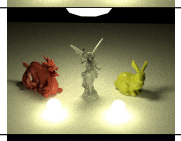
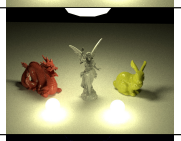
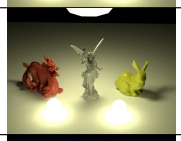
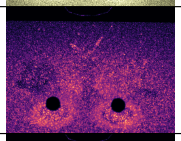
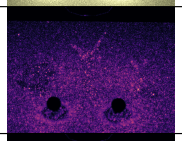
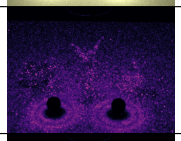
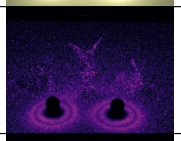
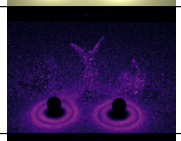
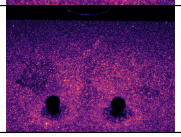
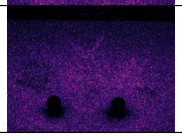
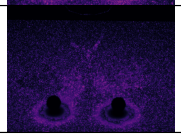
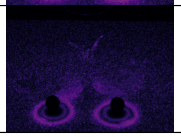
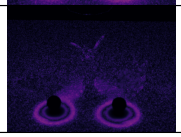
Figure 21: Left) Dielectric objects in different lighting scenarios in Cornell Box.

6.3 Results

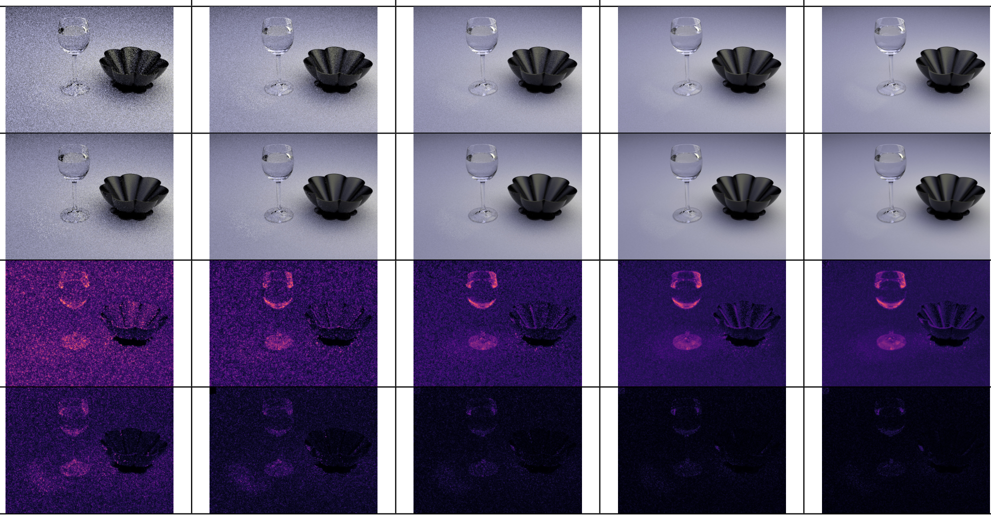
In order to create the reference images for all our tests. We decided to use the proxy algorithm (138). For each of the test scenes, we rendered 250 images, each with 16spp. After rendering those images, we averaged them and used the final solution as a reference image. It was observed that the final solution had much less noise, even compared to an image that was rendered with 1024spp. Following tables show rendered images with different samples per pixel (spp), first with brute force, then with next event estimation and then using multiple importance sampling with balance heuristic. Then for each of those images, we also get the output using ꟻLIP to visualize the difference with the reference image. We start using russian roulette after a path length of 3 for all methods (brute force, MIS and NEE). For outputs of ꟻLIP the closer the image gets to a black color, the closer it is getting to the reference image. A completely black output means that the image has no difference at all with the reference image.

	4	16	64	256	512
Brute force					
NEE					
MIS					
Brute Force (ꟻLIP)					
NEE (ꟻLIP)					
MIS (ꟻLIP)					

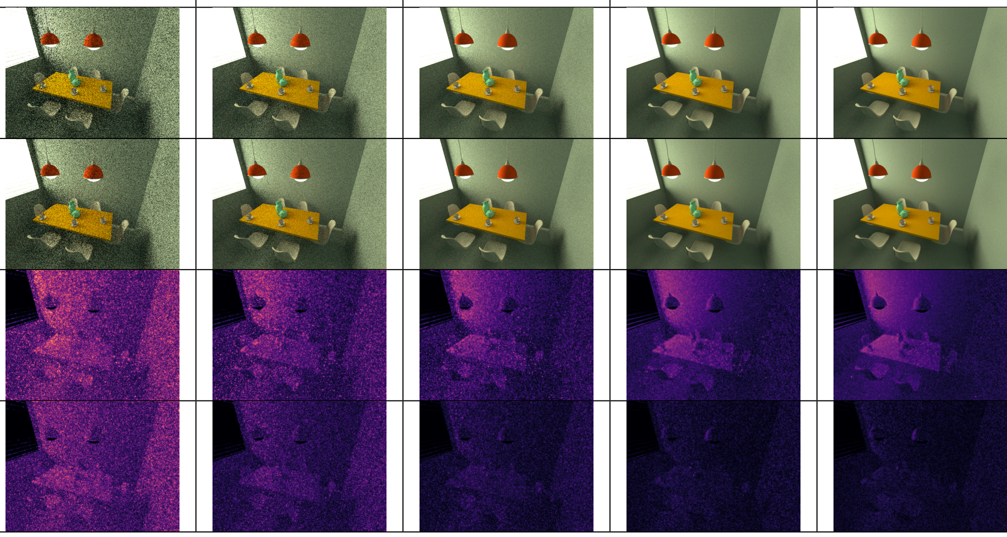
	4	16	64	256	512
NEE					
MIS					
NEE (ꟻLIP)					
MIS (ꟻLIP)					

	4	16	64	256	512
NEE					
MIS					
NEE (ꟻLIP)					
MIS (ꟻLIP)					

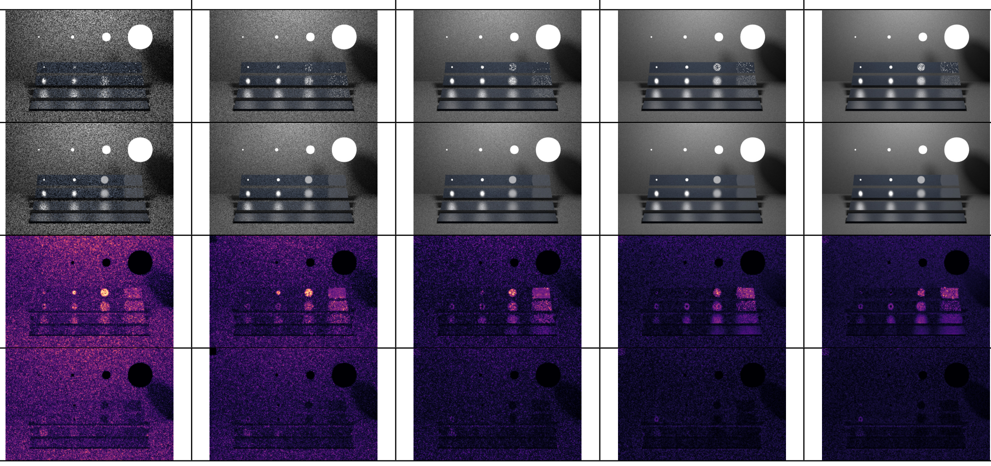
	4	16	64	256	512
NEE					
MIS					
NEE (ꟻLIP)					
MIS (ꟻLIP)					

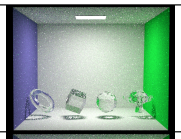
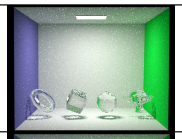
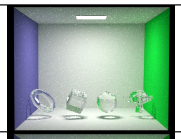
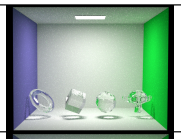
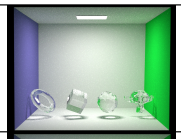
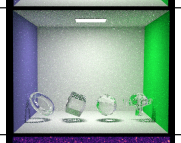
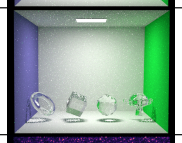
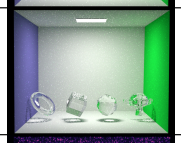
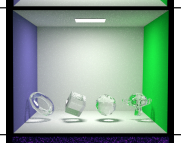
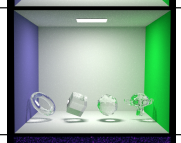
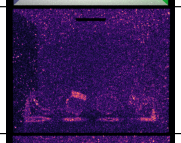
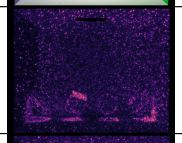
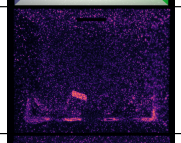
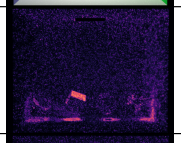
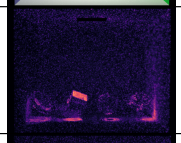
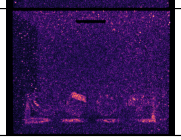
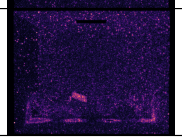
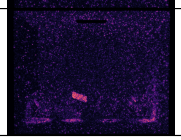
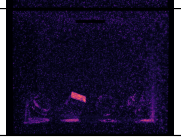
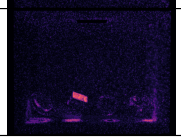


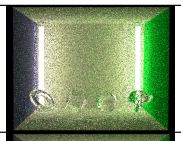
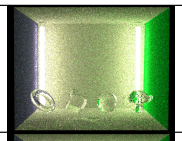
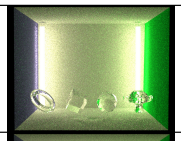
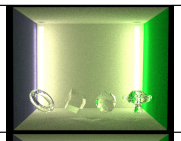
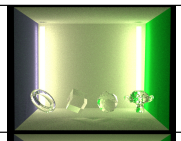
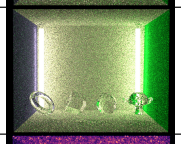
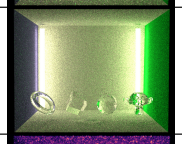
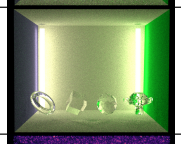
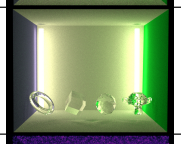
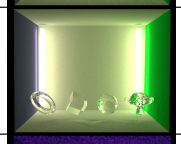
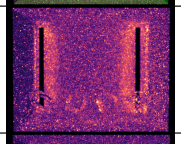
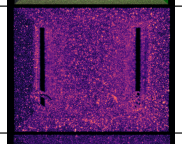
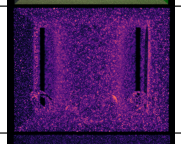
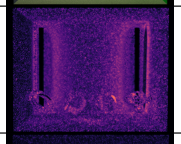
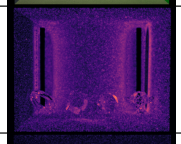
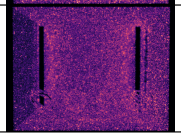
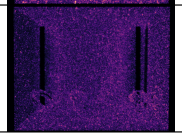
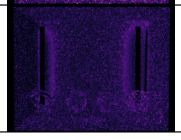
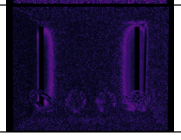
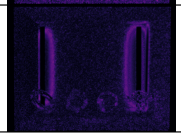
	4	16	64	256	512
NEE					
MIS					
NEE (ꟻLIP)					
MIS (ꟻLIP)					



	4	16	64	256	512
NEE					
MIS					
NEE (ꟻLIP)					
MIS (ꟻLIP)					



	4	16	64	256	512
NEE					
MIS					
NEE (ꟻLIP)					
MIS (ꟻLIP)					

	4	16	64	256	512
NEE					
MIS					
NEE (ꟻLIP)					
MIS (ꟻLIP)					

6.3.1 Discussion

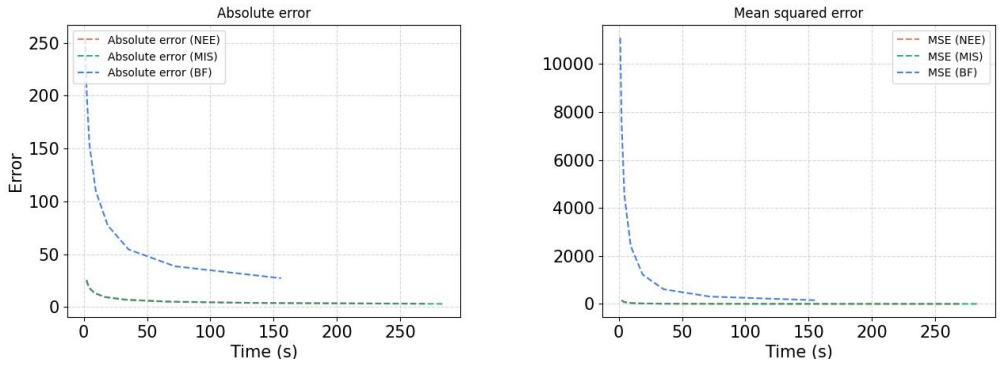


Figure 22: Absolute and Mean squared error on scene 1 for brute force, next event estimation and multiple importance sampling.

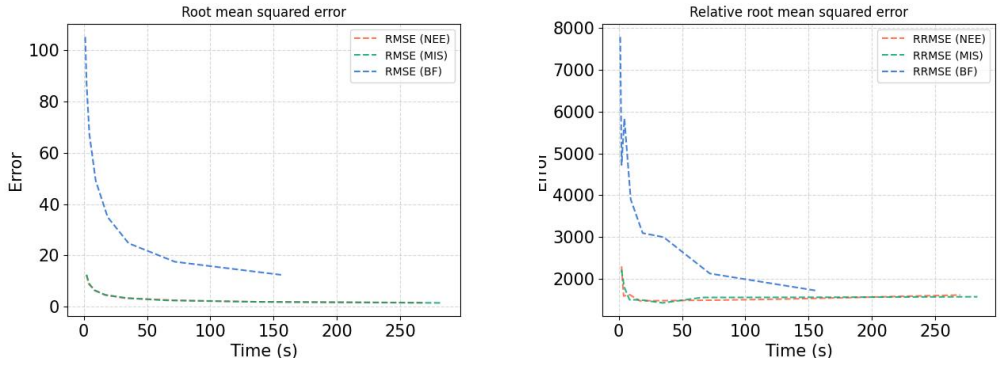


Figure 23: Root mean squared error and relative root mean squared error on scene 1 for brute force, next event estimation and multiple importance sampling.

We start our discussion with scene 1. It is no surprise to see that the brute force approach struggles to give good results unless a high amount of samples are taken. It can also be observed that the image difference generated by ꟻLIP for the brute force approach shows that even for 512 samples, there is still a lot of noise compared to the reference image. The time it takes to render the image is directly proportional to the amount of samples taken but it can be seen that the brute force approach is faster than both NEE and MIS. Starting with 4 samples per pixel, the RMS ϵ was calculated as 105 for the brute force approach but for NEE and MIS it is already on 12.4. At 512spp, the error using the brute force approach comes down to 12.4 which was the same amount of error for NEE and MIS when using only 4spp. Note the increased fluctuations in relative root mean squared error when using the brute force approach. This is because of the increased outliers (dead pixels) when using this approach.

Using NEE and MIS it can be observed that the time it took to render the image has increased. It is almost double the amount of time it took using brute force. It can be seen that the absolute error has reduced from a value of 254.5 using brute force to a value of 25.5 using NEE and MIS which is an improvement of a factor of 10x. But now it can also be seen that doubling the samples has diminishing returns i.e when going from 4spp to 8spp, the absolute error reduces to 18.3, and from 8spp to 16spp it reduces to 13.2. When using 512spp, the error was reduced to 3.05 using MIS but it took around 300 seconds compared to the 4spp render that took only 2 seconds. In other words, it is getting harder and

harder to reduce error further as the samples are increased. It is interesting to note that for this scene, the error is almost the same when using the same number of samples with MIS and NEE. Remember that multiple importance sampling balances different sampling techniques when in our case it is light sampling and BSDF sampling. But since all materials in our scene are diffuse, which is independent of the incoming and outgoing directions, the error that can arise due to BSDF sampling is low. This doesn't mean that we will always get the same result for NEE and MIS if only diffuse materials are used. This will be more clear in another scene we discuss later. The graphs for scene 2 are as follows:

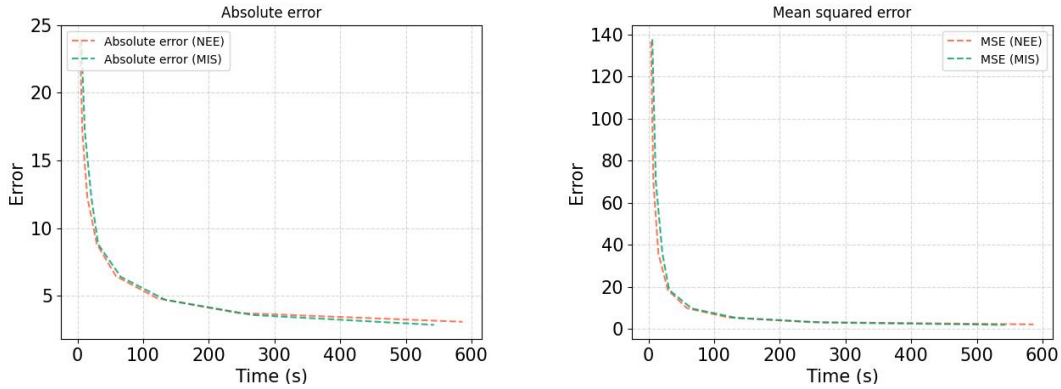


Figure 24: Absolute and Mean squared error on scene 2 for next event estimation and multiple importance sampling.

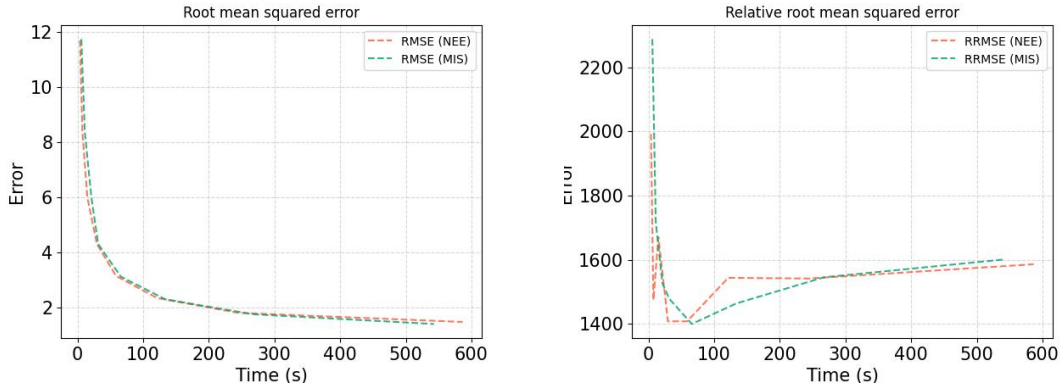


Figure 25: Root mean squared error and relative root mean squared error on scene 2 for next event estimation and multiple importance sampling.

For convenience, we have omitted the brute force approach for further analysis as it is already giving us poor results on even a simple scene like scene 1. For scene 2, a similar trend can be seen for both NEE and MIS for all errors except RRMSE. Both the methods now show a fluctuation of errors but NEE with a slightly higher rate. This shows that when the amount of objects increase in a scene as simple as the Cornell Box, MIS will give better results with less variance compared to NEE.

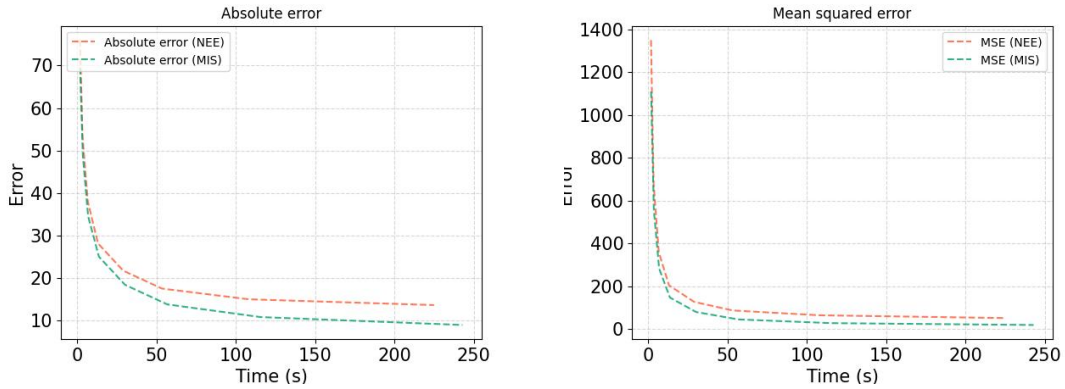


Figure 26: Absolute and Mean squared error on scene 3 for next event estimation and multiple importance sampling.

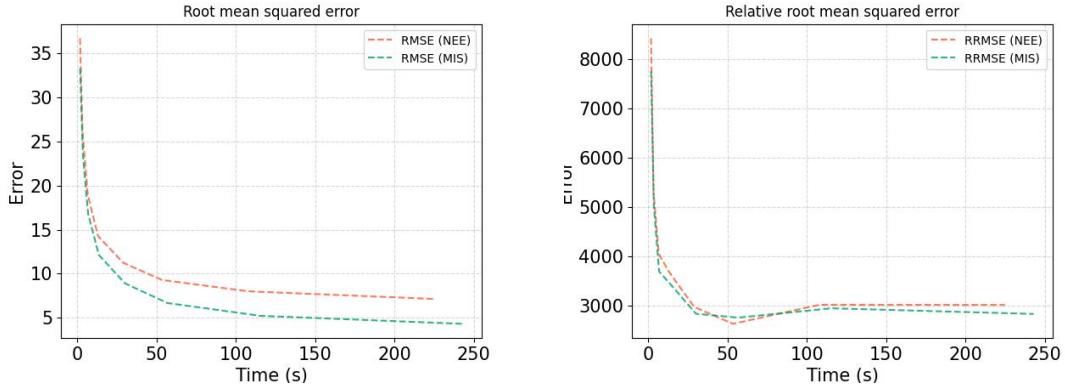


Figure 27: Root mean squared error and relative root mean squared error on scene 3 for next event estimation and multiple importance sampling.

On scene 3, it can be observed that NEE starts struggling to give comparable results as MIS. Even when using 4spp, the absolute error for NEE was 76 while it was 69.5 for MIS. At 1024spp RMSE was 7.1 for NEE while it was 4.3 for MIS, almost half as much noise in the resultant image.

The reason why NEE starts struggling on this scene is because this scene uses a smooth microfacet material for the 3 objects. This is different from a diffuse object where incident light is reflected equally in all directions. Here for a microfacet material, more light is reflected in some dominant directions (specular reflections) while less (or almost 0) light is reflected in other directions. As a result, this means that a majority of our paths would end up in low light (low radiance) and some minority of our paths would end up in high amount of light (high radiance). As an example, if 95% of paths end by hitting a non emitter object after several bounces, and the other 5% end at light with high radiance value, then we start experiencing a certain kind of noise known as **fireflies**. Fireflies in path tracing are bright pixels observed in our rendered image which are the result of these high radiance paths.

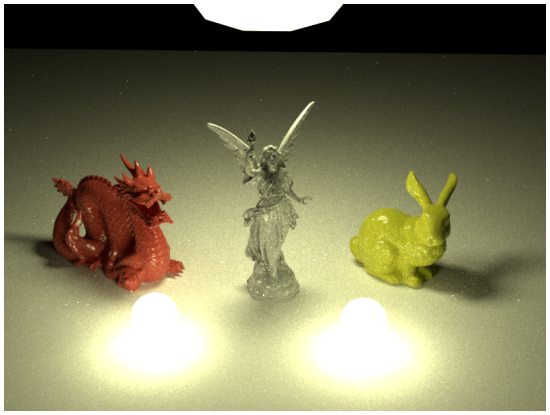


Figure 28: Notice small bright spots on this image rendered by next event estimation with 512spp.

While this could also happen with MIS in more complex scenes, it worked fairly well for our test scene since MIS handles this problem in a smarter way. All our path contributions are weighted using the balance heuristic and so whenever a path with high radiance is encountered, it gets divided by a high pdf value. A huge part of research in rendering is about finding ways to reduce fireflies in path tracing. One of the common approaches is clamping the high radiance value to a user provided threshold t [23]. Introducing clamping however, results in loss of energy i.e we will no longer converge to the exact solution. *Path regularization* [28] is another method to tackle fireflies in path tracing but also keeping the loss of energy as low as possible resulting in an unbiased image.

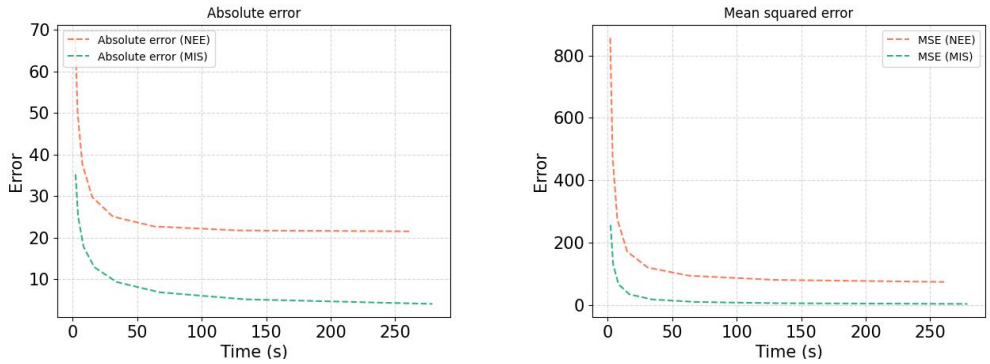


Figure 29: Abs. error and MSE on scene 4 for NEE and MIS.

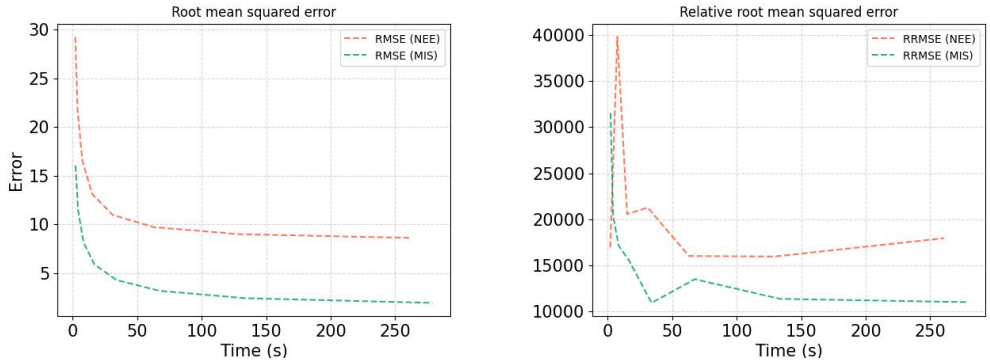


Figure 30: RMSE and RRMSE on scene 4 for NEE and MIS.

Looking at the graphs for scene 4, the error has a drastic difference when using both the methods. It can be seen that MIS is far better than NEE on this scene. We have an absolute error of 68 when using NEE and and error of 35.2 for MIS when using 4spp. At 512spp the error is down to 4 for MIS while it is almost stuck at 21.5 for NEE. In fact, the amount of error reduced from 256spp to 512spp for NEE was 21.7 and 21.5 respectively. A closer look at the images showed that images rendered by MIS were slightly dim for this scene while NEE was brighter with also significantly more fireflies at some locations. The image getting dimmer in the case of MIS also deviated the error for NEE since the reference image we use is an average of 250 MIS images with 16spp. The brighter NEE image is mostly because NEE struggles when using materials like a dielectric especially when **caustics** starts to appear. Caustics happen when light rays reflect and refract by a curved surface or object [43]. In our case, this was the dielectric glass in our scene with different refractive indices to model water in the glass. We will discuss this in another scene, how caustics can be a major problem when using path tracing and how can they be handled in a better way using another approach.

(a) 512spp with MIS (b) 512spp with NEE

Figure 31: Notice that the image rendered with NEE is slightly brighter. The glass itself is especially very bright at some places due to caustics.

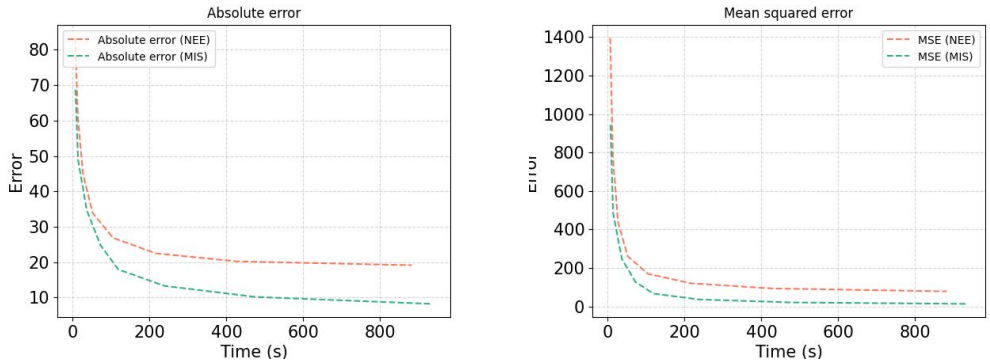


Figure 32: Abs. error and MSE on scene 5 for NEE and MIS.

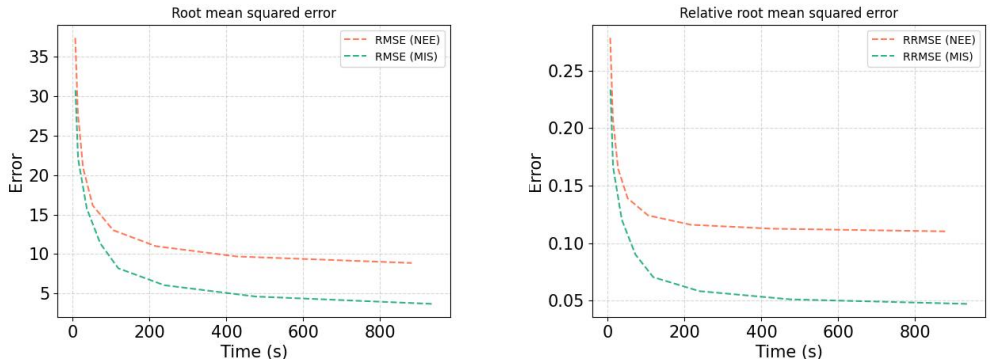
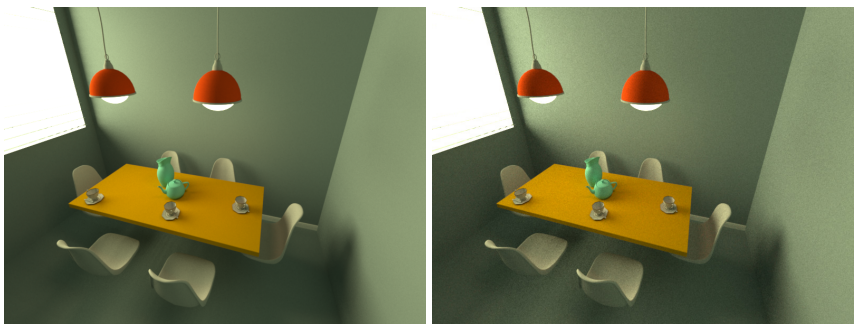


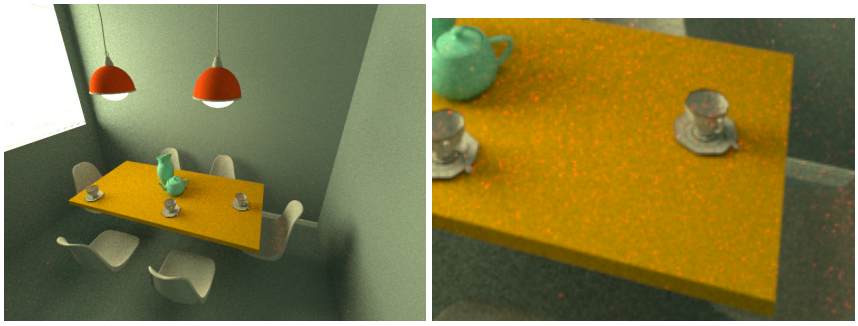
Figure 33: RMSE and RRMSE on scene 5 for NEE and MIS.

Scene 5 has all objects diffuse but this scene has an emitter with high radiance value outside the room and light comes into the room through a window. Two bulbs with a slightly lower radiance also serve as emitters. This scene is an example of how MIS can outperform NEE even for scenes that don't have objects with complex material properties. It can be observed that a lot of areas in the scene remain noisy when using NEE as it is difficult for light to reach those places. We have an RMSE of 8.86 when using NEE with 512spp while it is 3.68 when using MIS. It is a fairly common occurrence in path tracing that scenes with a very narrow opening of light source would result in noisy images. This is the reason why even using MIS with 512spp still do not give us a clear rendered image. Remember that we had an absolute error of 4 when using MIS with 512spp in scene 4 while here we have an error of around 8.2 with the same amount of spp. Bidirectional path tracing [29] is one way to greatly mitigate the noise that arises due to small/narrow light sources. Instead of creating one path from the viewpoint towards the light source, bidirectional path tracing as the name suggests, creates a path from the light source as well. Both these paths are then allowed to bounce around the scene collecting radiance until eventually they are connected at some point. This helps in reducing noise in scenes where it is difficult to hit a small emitter because we are also explicitly creating a path from the emitter itself. There was also a strange noise observed for images when using NEE which were not visible when using MIS. There were bright red pixels at different locations of the image. More of these dead pixels were observed near the table just below the emitters (lamps) hanging from the ceiling. Note that these lamps have a dark red covering. As a result, it is observed that if a bigger light source is very close to an object (lamp coverings in our case), explicitly doing emitter sampling can also result in noise like this. Our next scene explains this problem in more detail.



(a) Reference image (b) 512spp with MIS

Figure 34: The rendered image with 512spp using MIS still remains noisy.



(a) 512spp with NEE (b) Zoomed in

Figure 35: Notice the bright red pixels on the image rendered with NEE.

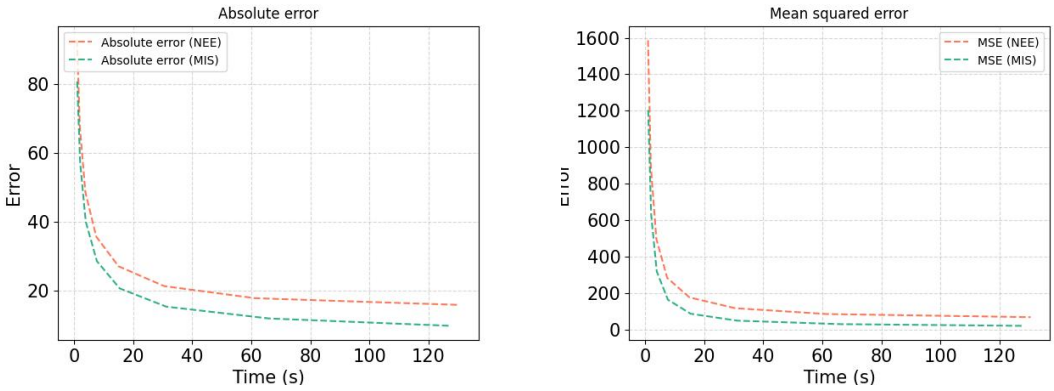


Figure 36: RMSE and RRMSE on scene 6 for NEE and MIS.

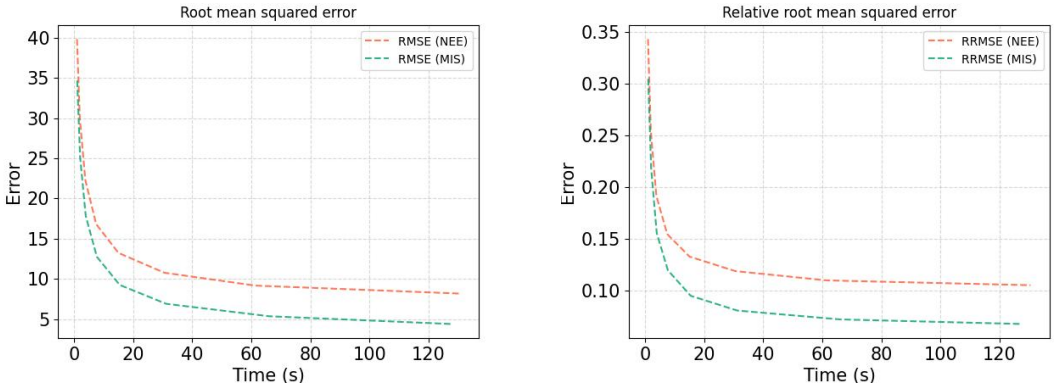


Figure 37: RMSE and RRMSE on scene 6 for NEE and MIS.

Scene 6 clearly demonstrates why using MIS is better than using NEE. Remember that the panels in this scene are mostly glossy at the top and almost rough at the bottom. In fact, it is visible on the rendered image that MIS is able to get the balance of both sampling techniques i.e BSDF and emitter sampling. When using NEE, It can be seen that as the size of the emitter increases the glossy panel close to the bigger light source is not sampled enough resulting in noise. This is because if the light source is small then sampling the light source gives better results as can be seen in the image rendered by NEE. But when the light source is large, and a glossy polished material is near to it, then sampling

the light source would result in high variance as the glossy material would only reflect the high amount of light close to it in dominant directions while other directions would be zero i.e when the BSDF has a large influence on the overall shape of the integrand then sampling only the light source leads to high variance. This will always happen for cases when the function we are approximating using monte carlo integration is large but the pdf we use to importance sample is low. A large function value divided with a low pdf value will generate spikes or *very large values*. MIS handles this efficiently because we weight the path contribution we are computing, with balance heuristic, balancing both the strategies (**??**).

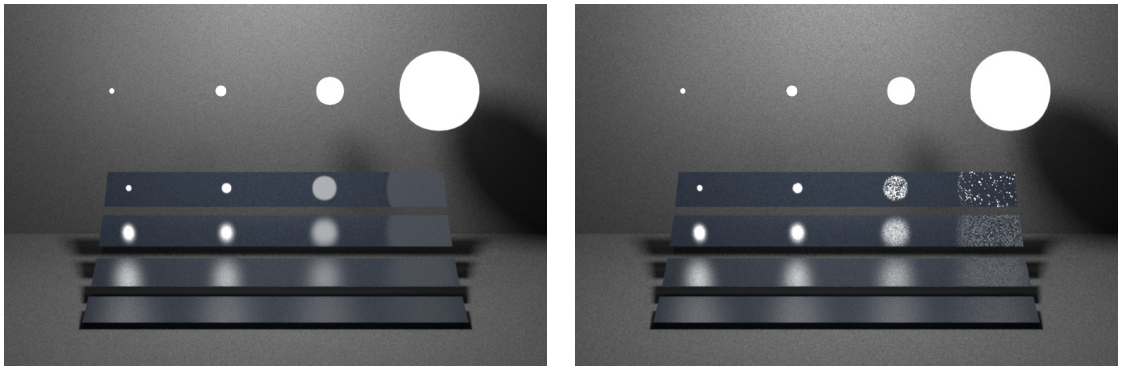


Figure 38: Left) Scene rendered with MIS (512spp). Right) Same scene rendered with NEE (512spp).

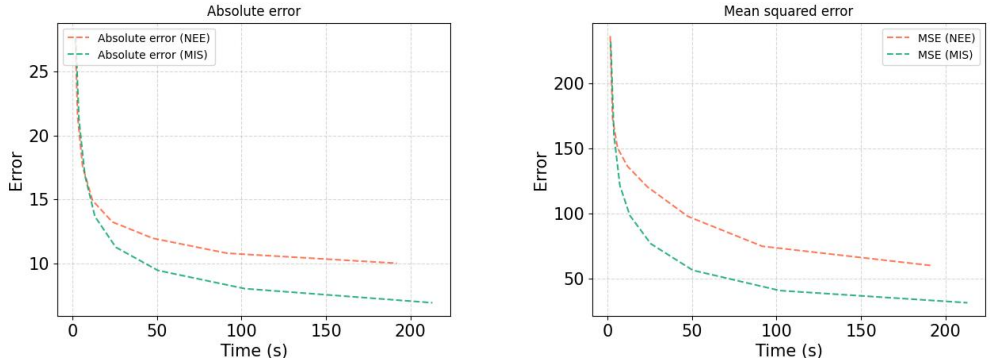


Figure 39: Abs. error and MSE on scene 7 for NEE and MIS.

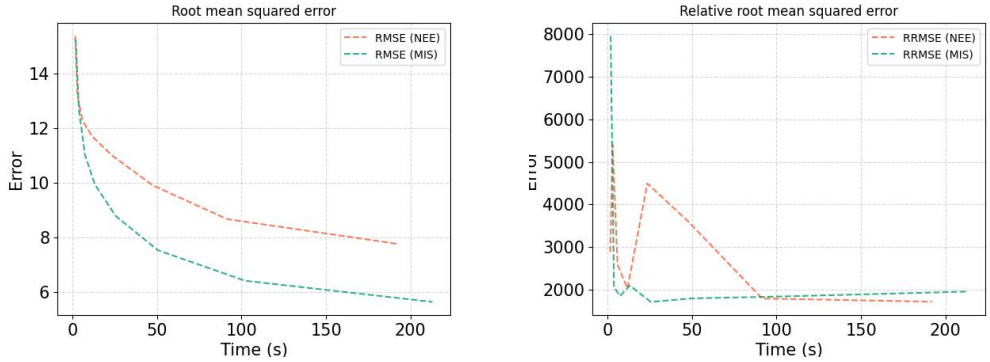


Figure 40: RMSE and RRMSE on scene 7 for NEE and MIS.

In scene 7, one thing to note is that both NEE and MIS appear to have the same amount of error at

the start. For 4spp, the absolute error NEE was 27.5 and for MIS it was 27.7, which is actually higher than NEE. A significant decrease in noise is noticed at a budget of 16spp. The absolute error is now at 17.6 for NEE and 16.9 for MIS. When using 512spp, we were down to an error of 10 for NEE but 7 for MIS.NEE has some spikes at some locations for the relative root mean squared error and this is because of the fireflies caused by dielectric objects. In fact, there are fireflies for MIS as well even at 512spp but significantly less than NEE.

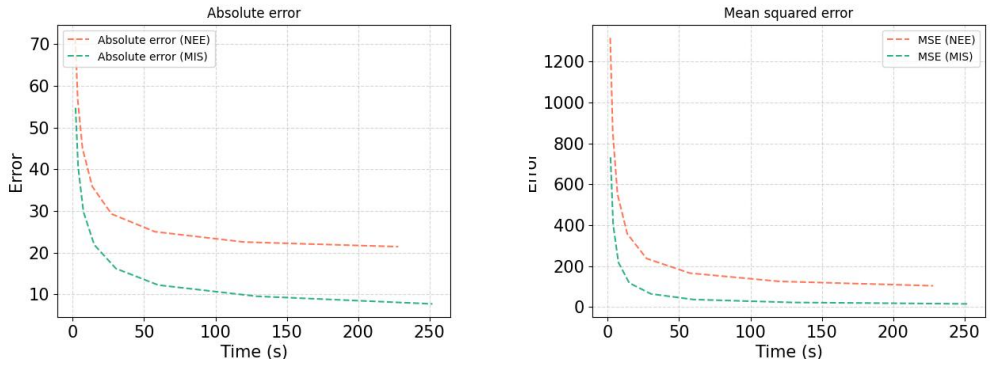


Figure 41: Abs. error and MSE on scene 8 for NEE and MIS.

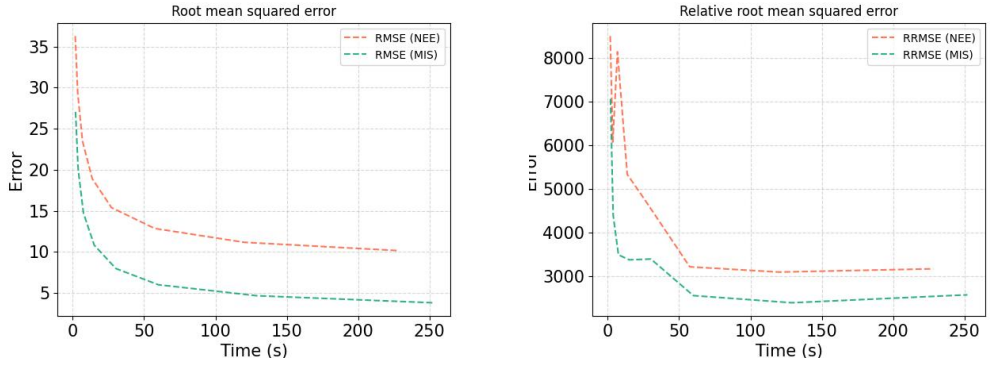


Figure 42: RMSE and RRMSE on scene 8 for NEE and MIS.

A similar trend as scene 7 can be observed in scene 8. Remember that this scene has exactly the same type of objects modeled as dielectric as scene 7. The only difference is the way the scene is illuminated with an area light source. In this scene, we have two cylindrical shaped area light sources in the two corners. It is observed that the error is significantly higher in this scene compared to scene 7. For 4spp, we start with an absolute error of 71.5 using NEE and an error of 54.7 using MIS. The error has more than doubled in the case of NEE. This is again because a light source in the middle of the scene lets more of the paths to hit the light source while for the case with two lights in the corners, it is difficult for the paths to move around the scene and eventually hit a light source. Most of the path would terminate close to the light source. This scene again has lots of fireflies for both NEE and MIS.
Photon Mapping [26] is a biased rendering algorithm that excels in scenes with glass materials and realizing noise free caustics. It is a two pass algorithm: in the first pass, we trace photons from light sources in our scene and cache them in a *photon map*. These photons have position $\boldsymbol{p}$, a direction $\boldsymbol{\omega_p}$ and photon power $\boldsymbol{\Phi}$. A photon's power is calculated as the total power of the light source divided by the number of photons emitted. In the second pass, we trace rays from the view point in our scene and approximate indirect illumination using the photons saved in our map.

6.3.1.1 Observing per pixel distribution

Let's discuss what the proxy algorithm tells us about the per pixel distribution of values in an image rendered by path tracing. We will use scene 3 and render it with both NEE and MIS. We will be rendering the scene with 4spp then 8spp and then 16spp. Afterwards we pick a pixel on the rendered image. For convenience, we show a reference image here so its easier to see which pixel we pick.

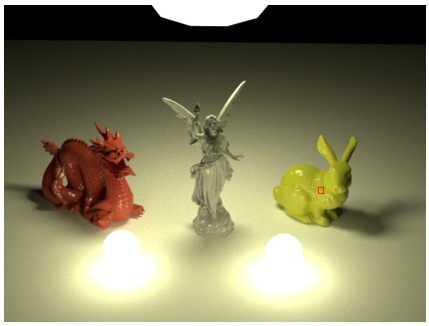


Figure 43: We pick a pixel on the yellow stanford bunny in scene 3.

Next, we render 250 images with the respective samples per pixel (4spp, 8spp, 16spp) using NEE and MIS. In Nori, the random number generator being used when sampling, is seeded according to the offset of the block of pixels, rendered in an image. This lets us render the same image every time we render it. But since we don't want that, we also multiply the offset with a random value seeded by time. For the rendered images, we pick the pixel we chose, and compute the signed absolute error ϵ as:

$$\epsilon = \hat{I} - I \tag{139}$$

For the result we get, we plot the histogram of error and error frequencies:

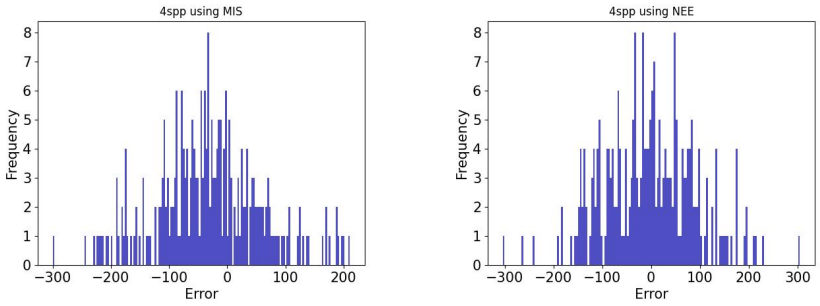


Figure 44: Histogram of 250 images rendered using MIS (left) and NEE (right) with 4spp.

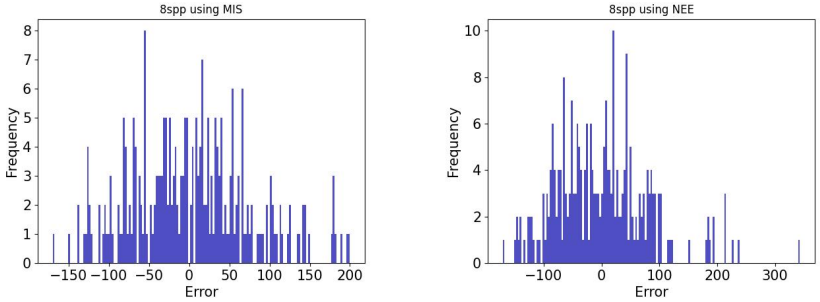


Figure 45: Histogram of 250 images rendered using MIS (left) and NEE (right) with 8spp.

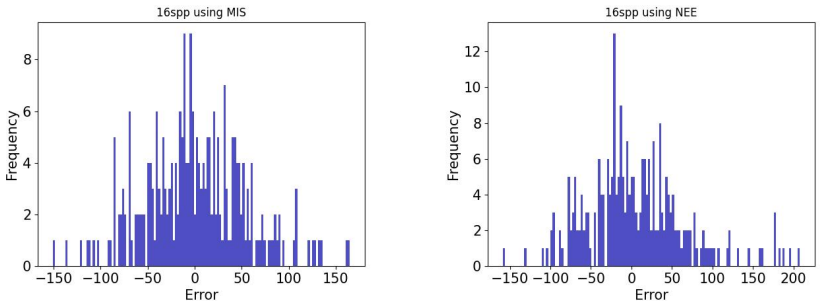


Figure 46: Histogram of 250 images rendered using MIS (left) and NEE (right) with 16spp.

Comparing histograms of MIS and NEE, it can be seen that the shape of the histogram is a normal distribution (CLT). Note that the range of errors for MIS when using 4spp is from -300 to 200 while it is from -300 to 300 for NEE showing that the error using MIS is less spread out i.e less variance. A similar trend can be observed when the samples are increased to 8spp. Also note that more and more images are getting closer to the mean 0 as the samples are increased, showing convergence. Let us test a caustic pixel in scene 8 and observe its distributions. The pixel we pick is shown below:

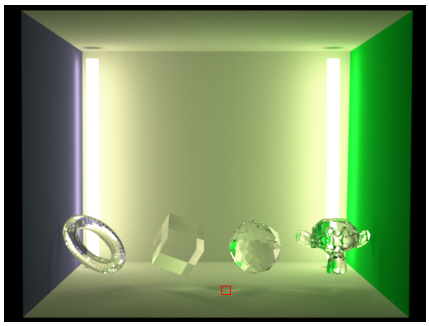


Figure 47: We pick a pixel near the caustic area in scene 8.

The histograms of 250 images rendered using NEE and MIS with 4spp, 8spp and 16spp are as follows:

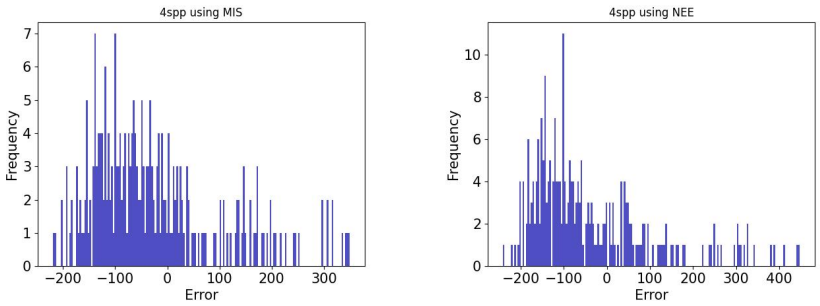


Figure 48: Histogram of 250 images rendered using MIS (left) and NEE (right) with 4spp.

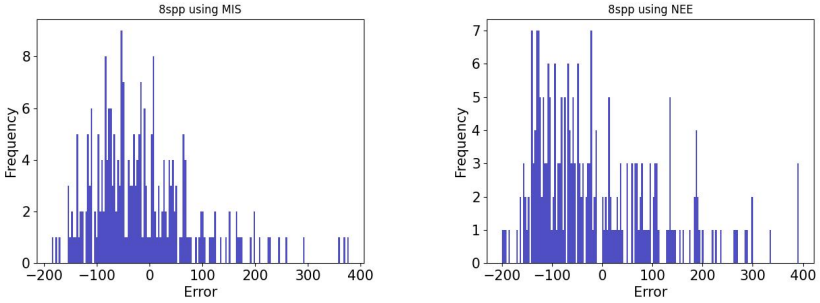


Figure 49: Histogram of 250 images rendered using MIS (left) and NEE (right) with 8spp.

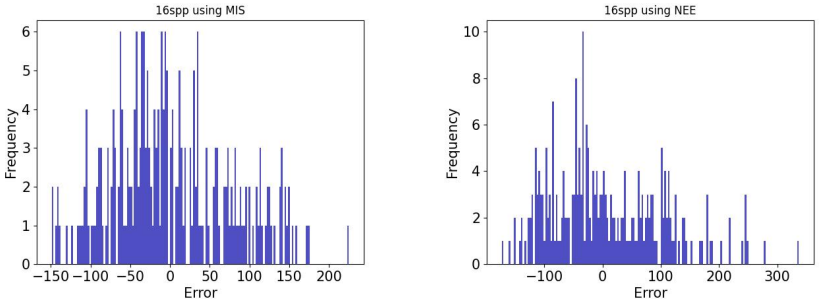


Figure 50: Histogram of 250 images rendered using MIS (left) and NEE (right) with 16spp.

Since this is a challenging scene even when using MIS, it can be seen that the histogram is more spread out. But it can be seen that more of the images are in the range of -150 to 0 for MIS than NEE at 4spp. As the samples per pixel are increased we start to observe that more images start to gather around the mean 0. But it can also be seen that a lot of images are still in the negative range. At 16spp, we can see that the range of the spread of error for the images has reduced from -200 to 400, to -150 to 250 for MIS. For NEE it is still -150 to 330. However it can be seen that both methods show convergence with more and more images having an error closer to the mean. We decided to count the number of images that lie under the range [-30, 30] for both methods and it was observed that when using 4spp MIS had 44 images while NEE had 23. At 8spp, MIS had 62 while NEE had 38. At 16spp, MIS had 74 while NEE had 55. Overall it can be said that since this scene is a bit more challenging because of the material being used, the variance still remains high compared to scene 3 where the error converged more quickly to 0.

6.3.1.2 Changing path length when using russian roulette

As a final test, we observe what happens if we increase the path length at which we start using russian roulette. Previously, we set this to 3 for all scenes tested. It has been observed that choosing a path length of 3 is more or less enough and works for most scenes but it can always be the case that some scenes might perform better with an increased path length. We pick scene 8 again and test it with path lengths 6 and 12 and observe if there's a significant reduce in error. The reason for picking scene 8 is 1) it performs poorly even when using higher spp for both MIS and NEE and 2) there's a possibility that increasing the path length may help reducing the error because of the way light sources are placed in this scene which might be a disadvantage when using a lower path length. All scenes were rendered using 512spp.

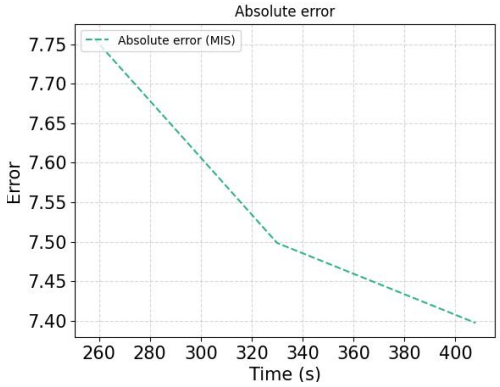


Figure 51: Absolute error for path lengths 3, 6 and 12 for scene 8.

It can be seen from the graph that the error reduces but not a lot. The error went down from 7.75 to 7.5 and then to 7.4. Whats worse is that the time it took to render the image increased from 258 seconds to 330 seconds for a path length of 6 and 408 seconds for a path length of 12. This is because now our paths survive much longer than they were before.

7 Conclusion

In this thesis, we started with a discussion of some important concepts that are relevant to implement a light transport algorithm known as path tracing. For our implementation, we first discussed a data structure that accelerates our intersection tests in our scene. Afterwards, we went through ways to transform our uniform sample on a square to different distributions. These transformations are crucial to sample different functions as we are using monte carlo integration to approximate the rendering equation. We then discussed how to model different types of materials in our scene.

Our first ray tracing technique was distributed ray tracing which only implements direct illumination in our scenes. We then implemented whitted style ray tracing which enables reflections and refractions in our scene, while also enabling multiple bounces of rays. Implementing these methods gave us the base to finally start with path tracing and after arriving at a monte carlo estimate for a single path in our scene, we implemented a brute force method that uses russian roulette to terminate our paths. Afterwards, we discussed a method known as next event estimation (NEE) which helps reduce variance in our images even further. We then coupled NEE with multiple importance sampling (MIS) to give appropriate weights to multiple sampling strategies in our estimator.

Analysis on scenes showed that MIS always perform either as good as NEE or better. It was found that NEE has its shortcomings especially in complex lighting situations or scenes with complex materials other than diffuse. We observed per pixel distributions and noticed that MIS converged faster than NEE as samples are increased. However, MIS also didn't perform that well in some scenes and the error was still high with noisy images. There are still many ways other than just using MIS to improve path tracing such as bidirectional path tracing [29] and photon mapping [26] which we discussed briefly. At the end we came to a conclusion that as scenes start getting more complex with different objects causing obstruction, complex lighting situations and varied materials, the error starts to increase with respect to our ground truth. And so, the topic of light transport simulation remains an open research in computer graphics with ongoing advancements as we discover new ways to solve the rendering equation.

References

[1] Matt Taylor (64.github.io). *Article on Tone Mapping by Matt Taylor.* 2019. URL: https://64.github.io/tonemapping/.

[2] Pontus Andersson et al. "ꟻLIP: A Difference Evaluator for Alternating Images". In: *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 3.2 (2020), 15:1–15:23. DOI: 10.1145/3406183.

[3] Wikipedia Andy Anderson. *Solid Angle Steradian.* 2023. URL: https://en.wikipedia.org/wiki/Solidangle#/media/File:SolidAngle,1Steradian.svg.

[4] Kavita Bala, Philippe Bekaert, and Philip Dutré. "Advanced Global Illumination Course Notes". In: (2002).

[5] Benedikt Bitterli. *Rendering resources.* https://benedikt-bitterli.me/resources/. 2016.

[6] Blender Online Community. *Blender - a 3D modelling and rendering package.* Blender Foundation. Blender Institute, Amsterdam, 2023. URL: http://www.blender.org.

[7] James F Blinn. "Models of light reflection for computer synthesized pictures". In: *Proceedings of the 4th annual conference on Computer graphics and interactive techniques.* 1977, pp. 192–198.

[8] Jakub Boksansky. *Crash Course in BRDF implementation.* 2021. URL: https://boksajak.github.io/files/CrashCourseBRDF.pdf.

[9] Brent Burley et al. "The design and evolution of disney's hyperion renderer". In: *ACM Transactions on Graphics (TOG)* 37.3 (2018), pp. 1–22.

[10] Adam Celarek. "Quantifying the convergence of light transport algorithms". PhD thesis. Wien, 2017.

[11] Adam Celarek et al. "Quantifying the error of light transport algorithms". In: *Computer Graphics Forum.* Vol. 38. 4. Wiley Online Library. 2019, pp. 111–121.

[12] Per Christensen et al. "Renderman: An advanced path-tracing architecture for movie rendering". In: *ACM Transactions on Graphics (TOG)* 37.3 (2018), pp. 1–21.

[13] Cambridge in Color. *Article on Dynamic Range in Digital Photography.* 2023. URL: https://www.cambridgeincolour.com/tutorials/dynamic-range.htm.

[14] Robert L Cook, Thomas Porter, and Loren Carpenter. "Distributed ray tracing". In: *Proceedings of the 11th annual conference on Computer graphics and interactive techniques.* 1984, pp. 137–145.

[15] Robert L Cook and Kenneth E. Torrance. "A reflectance model for computer graphics". In: *ACM Transactions on Graphics (ToG)* 1.1 (1982), pp. 7–24.

[16] Cornell. *Cornell box real photograph with rendered image.* URL: https://www.graphics.cornell.edu/online/box/compare.html.

[17] Keenan Crane. *Computer Graphics Lecture on The Rendering Equation by Prof. Keenan Crane.* 2020. URL: https://www.youtube.com/watch?v=Ttxdbn7TSLI&list=PL9jI1bdZmz2emSh0UQ5iOdT2xRHFHL7E&index=17.

[18] René Descartes. "La Dioptrique (1637)". In: *Œuvres et lettres* (1987).

[19] Philip Dutré and Yves D Willems. "Mathematical frameworks and Monte Carlo Algorithms for global illumination in computer graphics". PhD thesis. PhD thesis, Katholieke Universiteit Leuven, Belgium, 1996.

[20] Augustin Jean Fresnel. *Mémoire sur la double réfraction.* Vol. 7. Chez Firmin Didot, Père et Fils, Libraires, 1827.

[21] Iliyan Georgiev. “Path sampling techniques for efficient light transport simulation”. In: (2015).

[22] Cindy M Goral et al. “Modeling the interaction of light between diffuse surfaces”. In: *ACM SIGGRAPH computer graphics* 18.3 (1984), pp. 213–222.

[23] Eric Haines and Tomas Akenine-Möller. *Ray Tracing Gems: High-Quality and Real-Time Rendering with DXR and Other APIs*. Springer, 2019.

[24] Wenzel Jakob et al. “Dr.Jit: A Just-In-Time Compiler for Differentiable Rendering (Mitsuba 3 BSDFs)”. In: *Transactions on Graphics (Proceedings of SIGGRAPH)* 41.4 (July 2022). DOI: 10.1145/3528223.3530099. URL: https://mitsuba.readthedocs.io/en/stable/src/generated/pluginsbsdfs.html.

[25] Wenzel Jakob et al. *Nori, an educational ray tracer*. 2021. URL: https://wjakob.github.io/nori/.

[26] Henrik Wann Jensen. *Realistic image synthesis using photon mapping*. Vol. 364. Ak Peters Natick, 2001.

[27] James T Kajiya. “The rendering equation”. In: *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*. 1986, pp. 143–150.

[28] Anton S Kaplanyan and Carsten Dachsbacher. “Path space regularization for holistic and robust light transport”. In: *Computer Graphics Forum*. Vol. 32. 2pt1. Wiley Online Library. 2013, pp. 63–72.

[29] Eric P Lafortune and Yves D Willems. “Bi-directional path tracing”. In: (1993).

[30] Radoslaw Mantiuk et al. “Color correction for tone mapping”. In: *Computer graphics forum*. Vol. 28. 2. Wiley Online Library. 2009, pp. 193–202.

[31] Martin Newell. *Article on Creation of Utah Teapot*. 1975. URL: https://www.historyofinformation.com/detail.php?id=2449.

[32] Matt Pharr, Wenzel Jakob, and Greg Humphreys. *Physically based rendering: From theory to implementation*. Morgan Kaufmann, 2016.

[33] Elemental Ray. *Light Reflectance on Diffuse, Specular and Glossy materials*. 2013. URL: https://elementalray.files.wordpress.com/2013/01/dgs.png.

[34] Erik Reinhard et al. “Photographic tone reproduction for digital images”. In: *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*. 2002, pp. 267–276.

[35] Stanford. *Stanford Bunny*. 2021. URL: http://graphics.stanford.edu/data/3Dscanrep/.

[36] Stanford. *Stanford Lucy*. 1998. URL: http://graphics.stanford.edu/data/3Dscanrep/.

[37] Stanford. *Stanford XYZ Dragon*. -. URL: http://graphics.stanford.edu/data/3Dscanrep/.

[38] Matthias Teschner. *Advanced Computer Graphics Lecture notes - Solid Angle*. 2023. URL: https://cg.informatik.uni-freiburg.de/coursenotes/graphics201light.pdf.

[39] Eric Veach. *Robust Monte Carlo methods for light transport simulation*. Stanford University, 1998.

[40] Eric Veach and Leonidas J Guibas. “Optimally comning sampling techniques for Monte Carlo rendering”. In: *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*. 1995, pp. 419–428.

[41] Bruce Walter et al. “Microfacet Models for Refraction through Rough Surfaces.” In: *Rendering techniques* 2007 (2007), 18th.

[42] Turner Whitted. “An improved illumination model for shaded display”. In: *ACM Siggraph 2005 Courses*. 2005, 4–es.

[43] Wikipedia. *Article on caustics on Wikipedia.* 2023. URL: https://en.wikipedia.org/wiki/Caustic(optics).

[44] Wikipedia. *Article on geometric optics on Wikipedia.* 2022. URL: https://en.wikipedia.org/wiki/Geometricaloptics.

[45] Wikipedia. *Article on Luminance on Wikipedia.* 2023. URL: https://en.wikipedia.org/wiki/Luminance.