Monte Carlo methods for improved rendering

Tegan Brennan

Stephen Merity **Taiyo Wilson**

Abstract

Stochastic ray tracing is one of the fundamental algorithms in computer graphics for generating photo-realistic images. Stochastic ray tracing uses Monte Carlo integration to solve the rendering equation. This presents a major drawback however: noise. To decrease the visible noise to an acceptable level is computationally intensive.

In this project, we implement an unbiased Monte Carlo renderer as an experimental testbed for evaluating improved sampling strategies, both theoretically and practically motivated. We evaluate our results empirically, aiming to improve both the image quality and speed of convergence for our test images. Our results show that the improved sampling methods we use for rendering can give comparable image quality over twenty times faster than naive Monte Carlo methods.

1 Introduction

Raytracing is one of the most elegant techniques in computer graphics for producing realistic images of a given scene. The elegance comes from solving the light transport equation using Monte Carlo methods, which approximates the radiance of a given pixel by simulating how light paths could reach it in a physically accurate way. Many physical phenomena that are near impossible to compute with other techniques can be trivially modelled using a raytracer. To accurately render these scenes however is a computationally intensive task. For a high definition 1080p image, as might be rendered by Pixar, the radiance of over 2 million pixels need to be computed, requiring millions or billions of light paths depending on the complexity of the scene. Performing this naively is impractical, even with advanced hardware such as general purpose graphical programming units (GPGPUs) which can cast millions of rays per second. For the recent films *Cars* and *Monsters University*, each ray traced frame took between 15 and 24 hours to render (Christensen et al., 2006). We implement an unbiased Monte Carlo renderer and then evaluate various improved sampling strategies, both theoretically and practically motivated, to see the impact on both image quality and speed.

2 Approach

To simulate the global reflection of light throughout a scene, we need to consider how light moves around the scene. This is referred to as the light transport equation, described in Kajiya (1986). The light transport equation states that the radiance of a point is the sum of the self-emitted radiance (glow) and the reflected radiance. This value is then distributed according to the material's bidirectional reflectance distribution function (BRDF), which is detailed more in the next section. Intuitively, the BRDF looks at the angle of the reflected light and the outgoing angle to decide how much light should go in that direction. If the BRDF states the surface is perfectly mirror like, for example, only very specific angles make a contribution to the reflected radiance.

$$L_{out}(x, \theta_o) = L_{emit}(x, \theta_o) + L_{reflected}(x, \theta_o) \tag{1}$$

$$= L_e(x, \theta_o) + \int L_i(x, \theta_i) f_r(x, \theta_i, \theta_o) |\theta_i \cdot N_x| \partial w_i \tag{2}$$

$$= \text{emitted from object} + \text{reflected onto object} \tag{3}$$

The integral for the reflected light $L_{reflected}$ is computed over the hemisphere of all possible incoming directions. As this is impossible to compute exactly, it is approximated using Monte Carlo methods, turning the integral into a finite sum of samples. The primary issue that can occur is when noise is too strong to give reasonable estimations as to the radiance. Whilst the expected value remains accurate, the standard deviation (error) decreases by $\frac{1}{\sqrt{n}}$, where n is the number of samples. To achieve a noise free image requires a large number of samples, especially if the scene is complex.

To improve this, various sampling improvements can be used, aiming to only perform sampling in areas useful in determining the expected value. In the Methods section, we intermix both the description of the sampling improvement and the impact on our resulting renders.

2.1 Motivation for using Monte Carlo methods for rendering

2.1.1 Rendering as a high dimensional problem

Performing rendering via sampling is a high dimensional problem with many potential complications. Even beyond solving the light transport equation, there are many other opportunities for Monte Carlo methods in rendering. Solving the light transport equation itself involves computing the expected value over a two-dimensional hemisphere. Accurately modeling any physical phenomena results in adding additional dimensions to the problem. If we perform anti-aliasing (supersampling the pixels to prevent jagged edges), for example, we add another two dimensions to consideration. Other similar effects, such as depth of field or motion blur, again mean averaging over an additional set of dimensions.

Without Monte Carlo methods, incorporating each of these can be a complex and computationally intensive problem. With Monte Carlo methods, all of these cases can be implemented with relative simplicity, frequently requiring few modifications to the base rendering algorithm.

2.1.2 Accuracy of rendering

As opposed to many other uses of Monte Carlo methods, rendering requires relatively high accuracy. The final output of the renderer is frequently presented to humans for viewing purposes without any additional modification or processing. Without the proper number of samples, renders commonly resemble vintage photographs.

Given this, one might fear that noise in the image would make the images aesthetically unusable, especially due to the high dynamic range of the human eye and the slow convergence of Monte Carlo methods in high dimensional domains. This is resolved in two separate ways. First, whilst humans have high dynamic range, they have low absolute sensitivity, meaning that small changes in pixel values are imperceptible. Second, the quality of a raytraced image can be improved progressively as required. If the image is still too noisy, rendering can continue from that point, using more samples to reduce the error, with the only penalty being that the image takes longer to render.

2.2 Test scenes

When testing our raytracer and the various sampling methods, we primarily focus on rendering variations of the Cornell box which was created as a physical model in 1984 to serve as a base case for various computer graphics tests. The Cornell box scene appears in many different variations[1], depending on what is being tested. Our scenes replace the boxes with spheres as our raytracer currently only supports sphere primitives and we wanted to focus on the sampling methods rather than the mechanics behind raytracing.

[1]For various other scenes, see Henrik Wann Jensen's `http://graphics.ucsd.edu/~henrik/images/cbox.html`

3 Improved Monte Carlo methods

3.1 Importance sampling using materials via BRDF sampling

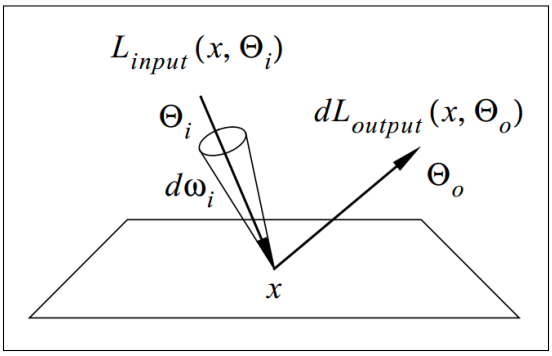
When light hits a surface, it bounces away from it unless completely absorbed. Working out the probability that it bounces in a given direction is a core question in accurate rendering. On some surfaces, such as perfect mirrors, the direction is highly defined, deviating minimally from the law of reflection[2]. As the surface becomes more diffuse however, such as a glossy imperfect mirror or a piece of paper, the direction becomes more and more widely distributed.

The reflectance properties of an object are governed by the bidirectional reflectance distribution function (BRDF) f_r of that object's material. The BRDF for a given real world object can be measured using a goniophotometer, though empirical approximations are used for most computer graphics applications. To calculate the BRDF, we use a given surface point x, the normal of the surface N_x, the incoming direction θ_i, and the outgoing direction θ_o. Formally, this is defined as:

$$f_r(x, \theta_i, \theta_o) = \frac{\partial L_{output}(x, \theta_o)}{L_{input}(x, \theta_i)|\theta_i \cdot N_x|\partial w_i}$$

Note that $|\theta_i \cdot N_x|$ is equal to the cosine angle between the two vectors.

Intuitively this maps out to:



Generating uniform random bounce directions is not the best method for approximating the integral if we have an idea about the behavior of the integrated function. On some surfaces, such as a glossy imperfect material, it makes the most sense to sample directions around the specular direction (in other words, the mirror reflection direction), as most of the reflected light originates from these directions. By generating sample rays according to the exact probability density function that defines the object's material, we are only sampling rays in the exact proportion to which their contributions matter.

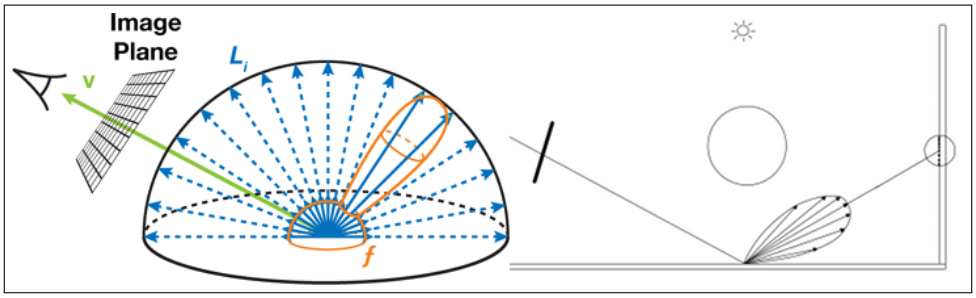


Figure 1: Uniform sampling of the potential bounce directions (blue) versus importance sampling using the object's bidirectional reflectance distribution function (orange).

[2]The law of reflection states that when a ray of light reflects off a surface, the angle of incidence is equal to the angle of reflection.

3.2 Importance sampling using explicit light sampling

In an unbiased Monte Carlo renderer, the light path being sampled must hit a luminaire (light source) before it can make any contribution to the radiance of a given pixel. When the luminaire is small, the probability that a random light path will intersect with it is highly unlikely, requiring a larger and larger number of light paths to be sampled, most of which contribute nothing to the radiance. These small but high intensity luminaires are not uncommon in computer graphics, with the most common example being the sun, which although small dominates the light contribution to most scenes.

By exploiting our knowledge of the scene, specifically the location of these luminaires, we can perform importance sampling towards the location of the light sources (Shirley et al., 1996). To do this, we select a random point on any surface that emits light (luminaire). We then check whether the given random point is visible from the hit position. If it isn't, the light source is not visible, resulting in no light contribution. If the random point is visible, we compute what portion of the light is reflected towards the outgoing light path according to the BRDF function.

Selecting a random point is important. Imagine sampling from a sphere. If a single point was used, such as the center, all shadows generated by that luminaire would be perfect hard projections of the shapes. If multiple points are used, evenly distributed over the sphere, the shadows would have a penumbra (partially shaded outer region of a cast shadow) related in size to the sphere. Taken over an infinite number of samples, and assuming that the selected point from the luminaire was generated truly at random, each of the points on the luminaire would be sampled evenly. When there are multiple luminaires in the scene, we can select to check for illumination from a single point from each of them or a single point from a randomly chosen luminaire. This has performance implications in certain scenes. For the full mechanics and modifications required when sampling from one or many luminaires, refer to appendix A.1.

Adding explicit light sampling additionally changes the way in which the light transport equation works. On the first bounce of the light path where the path is directly from the image plane to the scene ($depth = 0$), the light transport equation remains the same:

$$L_o(x, \theta_o) = \text{emitted from object} + \text{light from luminaires} + \text{reflected onto object}$$

This ensures that the first bounce, those that feed directly into the image, register light sources seen directly from the virtual camera as light sources. For all subsequent steps ($depth > 0$), the light transport equation drops the first term:

$$L_o(x, \theta_o) = \text{light from luminaires} + \text{reflected onto object}$$

This is required as otherwise double counting would occur. For a full description of why double counting occurs, refer to appendix A.2.

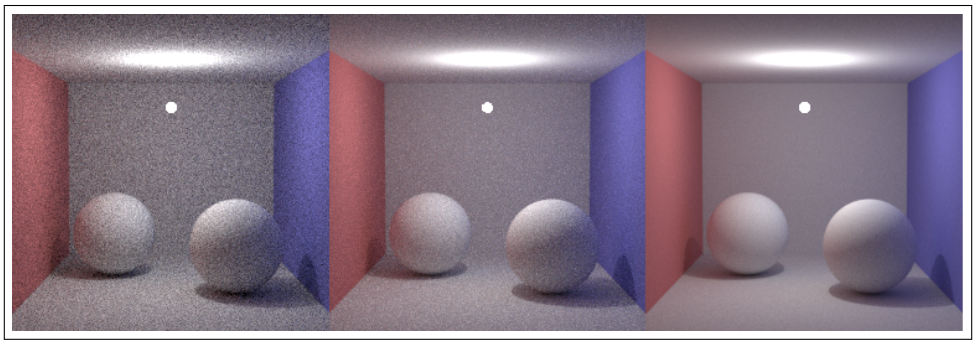


Figure 2: Impact of emitter sampling on speed and image quality. The image on the left used implicit emitter sampling and was run for 20,000 iterations, taking 2,583 seconds. The images on the middle and right used explicit emitter sampling and were run for 50 and 500 iterations. They took 9 and 101 seconds respectively, both resulting in a far higher quality image.

3.2.1 Rejection sampling: Russian roulette path termination

In a physically accurate model, an extremely bright light path may continue for hundreds or millions of bounces. This presents a practical problem, as computing such a long path is intensive and contributes little. If we truncated light paths after a fixed depth however, we would be introducing bias into the Monte Carlo approximation.

A theoretically more sound technique is Russian roulette path termination, which introduces a path termination probability q at each step. At each bounce in the light path, we stop traversing and return zero with probability q, and continue sampling as usual otherwise. As it does not cut off any of the subsequent paths in a deterministic way, but instead in a probabilistic way, we can compensate for the probabilistic truncation, yielding an unbiased estimator. In our implementation, the probability q is modified by the reflectance properties of the material currently being sampled. This results in a high probability of termination if the light path hits a black surface (i.e. most of the light contribution will be absorbed) and a low probability of termination if hitting a white surface.

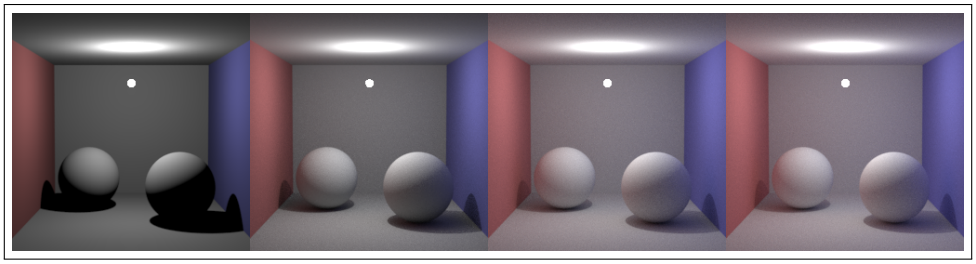


Figure 3: Impact of Russian roulette sampling on speed and image quality. The first three images have a maximum allowed light path length of 1, 2, and 3, taking 18 seconds, 31 seconds and 58 seconds respectively. The image on the right uses Russian roulette path termination, taking 83 seconds. Extreme bias can be seen in images of maximum allowed light path length of 1 and 2. In more complex scenes, even max depths of 10 can result in extreme bias.

3.2.2 Variance reduction and adaptive sampling

Monte Carlo raytracers traditionally work by taking a specified number of samples for every pixel in the image. Though this is guaranteed to produce a desirable result for large enough n, it can lead to substantial computational effort being spent on pixels that do not require any additional samples.

Some pixels are more noisy than others. The history of samples for that pixel may display a much higher variance when compared to, for example, a completely black pixel. It would thus make sense to spend computational time on pixels with high variance than those with low variance, for which we do not need any improvement.

To perform this variance reduction, we use adaptive sampling. Adaptive sampling is a method which seeks to reduce variance in the image by increasing the number of samples that pixels of high variance receive.

One of the issues with adaptive sampling is how to choose a robust adaptive refinement criterion. We are currently in the process of considering and experimenting with various methods to determine when the variance of a pixel has fallen below an acceptable threshold. The most straightforward implementation would be to simply keep track of all the samples for a pixel and to draw a new sample with probability proportional to the variance of the previous samples. In this way, we sample with higher probability from pixels whose histories show a higher variance.

In addition to per pixel considerations of variance, we can also consider the variance over neighboring pixels. For most images, we expect that nearby pixels would tend to demonstrate similar variance tendencies. Even if they don't, we wish to minimize this variance as much as possible, preventing alias issues which are prevalent in computer graphics. In order to take this into consideration, we can additionally consider how similar the current value at a pixel is to those that surround it and choose to sample from that pixel with probability proportional to this value.

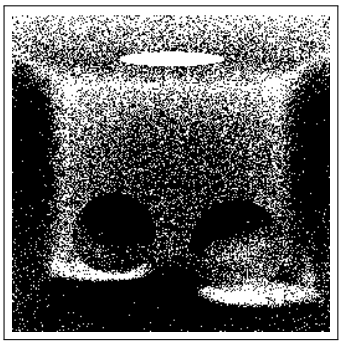
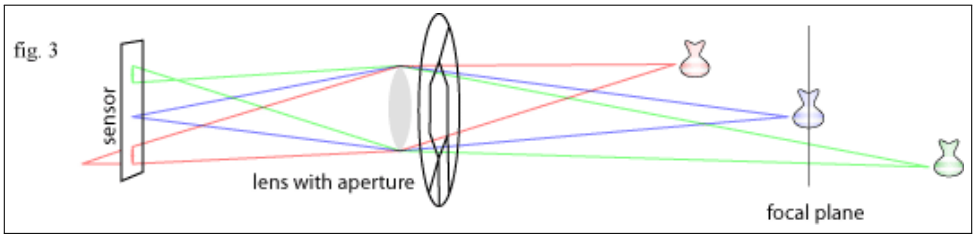


Figure 4: The accompanying image displays the relative variances of the pixels in the image, with pixels of higher variance colored darker than those of lighter variance. Here the variance was measured solely based on the history of the individual pixels.

As mentioned above, we are currently in the process of experimenting with both the above methods in order to find an effective measure of the variance at a pixel. As such, we do not have final results to compare using this method but have included the below image in order to demonstrate where the areas of high variance lie in our image.

3.2.3 Rendering realistic lenses with Monte Carlo

Monte Carlo rendering methods allows for trivial emulation of physical phenoma, including the behaviour of camera lenses. We chose to implement features such as aperture size and focal length in our raytracer to demonstrate the flexibility of our solution. The following figure shows a basic diagram of how the rays first enter the scene when hitting a camera lens.



Rays originate at the aperture, a small hole through which light enters. If the aperture is small, the rays are highly collimated (collimated light rays are near parallel); if it is large, the rays are more spread out. If the aperture is large, the field of view will be larger, and vice versa. Figure 5 shows the Cornell box with our standard aperture size (centered) as well as smaller apertures with flatter fields of view (left) and larger apertures with wider fields of view.

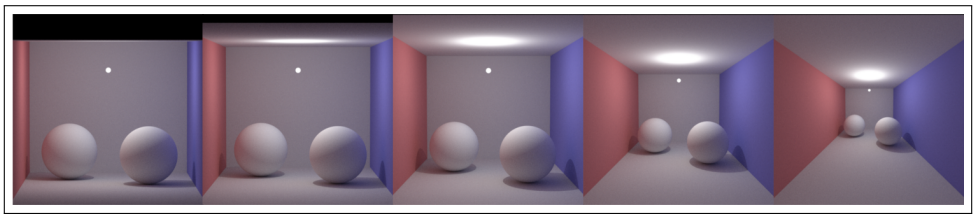


Figure 5: Impact of aperture size on field of view. The third image is taken using our standard aperture size. The first and second images have aperture sizes of $\frac{1}{4}$ and $\frac{1}{2}$ of the original, and the fourth and fifth images have aperture sizes of 2 and 4 times the original size, respectively.

From the aperture, the rays then pass through the image plane, which is where the pixel information is gathered. As the rays travel towards the image, the rays for each pixel converge at a certain point, called the focal point. The distance from the camera to this point is called the focal length. Images at the focal point will be in focus, while images in front of or behind this point will be blurred.

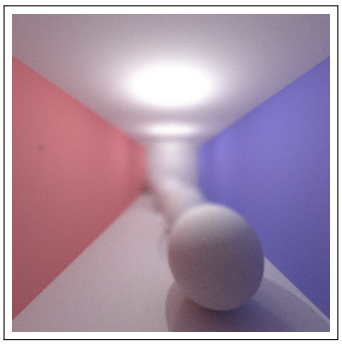


Figure 6: 'Cornell hallway' showing the effect of a shallow depth of field and short focal length. Only the front of the first ball is in focus and the rest of the scene is blurred increasingly past this point.

We accomplished this using Monte Carlo by randomly jittering the origin of each ray at each pixel. The direction of each ray is then calculated so that all of the pixel's rays converge after traveling a distance equivalent to the focal length. The aperture and the focal length are related by the cameras depth of field, which is the distance between the nearest and farthest objects in a scene that appear in focus. For smaller apertures, the rays are collimated, or less spread out, meaning the images are fairly well resolved even far from the focal point, resulting in a larger depth of field. The opposite is true of larger apertures, which have a very narrow depth of field. In Figure 7, we have created a scene of a hallway of balls, employing a large aperture size and short focal length. As a result, only the front of the very ball is in focus, and the rest of the image is blurred.

4 Discussion

4.1 Complications from Metropolis Light Transport

We originally intended to implement Metropolis Light Transport (MLT), as described in Veach and Guibas (1997), an algorithm which uses Metropolis-Hastings sampling to explore the space of light paths, biasing towards light sources to effectively approximate the light rendering equation with fewer samples. In contrast with a traditional Monte Carlo raytracer, the MLT algorithm works not by firing a new ray into the image for each new sample but by carefully mutating a currently existing light path to produce a sequence of paths. These mutations must be carefully constructed in order to ensure a low variance, low auto-correlation, stratified sequence of samples with a high acceptance rate. Many methods have been presented to handle bias occurring due to mutations, including Kelemen et al. (2002) and Cline and Egbert (2005), but all of them require extreme complexity and care. If they are properly implemented however, Ashikhmin et al. (2001) shows that the asymptotic time complexity is no worse than uncorrelated Monte Carlo methods such as path tracing.

When we initially worked on MLT, much of our time was focused on the mutation functions and our acceptance probabilities, paying special attention to the Hasting term of the Metropolis-Hastings acceptance ratio. Doing so requires computing the probability of generating the new mutated path from our starting path, which includes computing both the probability of deleting the deleted subpath, $x_s \dots x_t$, from the starting path as well as the probability that the vertices of the new path would be chosen to replace them. The latter additionally includes computing all the ways in which the new vertices could split between subpaths from x_t to x_s. This mathematically intensive task is a cause of frustration even for experts (Hoberock and Hart, 2010) and we eventually decided that it would be too intensive and error prone to implement in the time we had available.

4.2 Implementation and code

The renderer we developed, Montelight, was implemented from scratch in C++. We initially intended to write Montelight in Python, allowing for rapid prototyping and a focus on the stochastic optimizations rather than engineering, but as features were added, and larger experiments attempted, the slow speed of Python became a problem. We attempted to rectify this by using PyPy, a fast, compliant alternative implementation of the Python language. Whilst PyPy was 15 times faster than standard Python, it was still too slow compared to a low level implementation in C or C++. Re-implementing the project in C++ gave us an additional 16 times speed boost over PyPy. We believe

this shows that high level languages, such as Python or R, are not appropriate for solving computationally intensive problems using Monte Carlo methods, and that other speed optimized languages such as C, C++, or Julia should be used.

The source code available at `https://github.com/Smerity/montelight-cpp` includes instructions for both compiling and running the program. Other than requiring a recent C++11 compiler, such as g++ or clang, our renderer does not use any other libraries or extensions.

4.3 Future Work

This work could be extended to consider a large number of interesting problems, both in modeling complex physical phenomena efficiently and also improving the speed our the raytracer. Numerous engineering optimizations, such as octrees for ray-intersection calculations or complex surface rendering, were ignored to keep focus on the stochastic optimization aspect. More complex materials could also be implemented, including mirrors, metallic surfaces, and dielectric surfaces such as glass.

5 Conclusion

In this project, we implemented a stochastic raytracer to serve as a platform for experimenting with various Monte Carlo methods. Using importance sampling, variance reduction, and rejection sampling, we were able to substantially improve both the speed and quality of our rendered images, whilst not compromising quality. Our results show that the improved sampling methods we use for rendering can give comparable image quality over twenty times faster than naive Monte Carlo methods without restricting the scenes that can be tackled.

References

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

Per H Christensen, Julian Fong, David M Laur, and Dana Batali. Ray tracing for the movie Cars. In *Interactive Ray Tracing 2006, IEEE Symposium on*, pages 1–6. IEEE, 2006.

David Cline and Parris Egbert. A practical introduction to Metropolis Light Transport. Technical report, Tech. rep., Brigham Young University, 2005.

Jared Hoberock and John C Hart. Arbitrary importance functions for Metropolis Light Transport. In *Computer Graphics Forum*, volume 29, pages 1993–2003. Wiley Online Library, 2010.

James T Kajiya. The rendering equation. In *ACM Siggraph Computer Graphics*, volume 20, pages 143–150. ACM, 1986.

Csaba Kelemen, László Szirmay-Kalos, György Antal, and Ferenc Csonka. A simple and robust mutation strategy for the Metropolis Light Transport algorithm. In *Computer Graphics Forum*, volume 21, pages 531–540. Wiley Online Library, 2002.

Peter Shirley, Changyaw Wang, and Kurt Zimmerman. Monte carlo techniques for direct lighting calculations. *ACM Transactions on Graphics (TOG)*, 15(1):1–36, 1996.

Eric Veach and Leonidas J Guibas. Metropolis Light Transport. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 65–76. ACM Press/Addison-Wesley Publishing Co., 1997.

A Details on explicit light sampling

A.1 Efficiently sampling many luminaires

If one random point is sampled from each of the luminaires in the scene, no reweighting is necessary to keep a proper distribution. A key question is whether we can select how many of the luminaires to be sampled at each stage of the light path. This can be useful as sampling $|luminaires|$ rays at each stage of the light path can be computationally intensive, especially when there are a large number of luminaires, of which many of these luminaires may not be visible at all. We might find that we can save computational time by only sampling one randomly selected luminaire at each stage of the light path and instead performing more samples. Care must be taken to ensure we do not skew the expected mean. This is as, when sampled to infinity, each of the luminaires must be sampled evenly, and the expected mean must be equal to what it would have been otherwise. If we select only one of the luminaires randomly, the light contribution must be multiplied by $|luminaires|$, as the probability of sampling from that luminaire would be $\frac{1}{|luminaires|}$.

Intuitively, we can model this as a problem where we wish to find the expected mean of a random throw of two independently weighted dice. The standard method would be to take n samples, where a sample is throwing both the dice, sum the scores, and then compute the mean over those samples. Imagine however that we could only throw one dice at a time, and that the dice was selected randomly. If we could only throw one dice at a time, with the dice randomly chosen, we could still compute the expected mean when both dice are thrown by taking $(sum(samples) \times 2)/n$.

A.2 Double counting

In order to prevent double counting of any luminaires, we must remove the first term of the light transport equation when performing explicit light sampling. The reason for this is not intuitively obvious but can be reasoned through.

Imagine a scenario in which we want to compute the radiance at a given point x and there is a single luminaire l visible from that point. If we perform explicit light sampling, we have the contribution from l to x. If we perform the exact integral over the hemisphere of reflected light, we also consider the contribution from l to x, as one of the reflected rays is the exact computation performed via explicit light sampling. This means that the contribution $L_{emit}(x, \theta_o)$ is added by both explicit light sampling and the integral of the reflected light. To maintain the correct solution, one of these must be removed. This is most effective when removing the first term of the light transport equation as importance sampling of luminaires via explicit light sampling will be far more efficient than calculating the equivalent sum via reflected light.