Computing Photorealistic Images: An Overview of Monte Carlo Global Illumination Algorithms

SEAN SIDDENS, University of California, Santa Cruz

Fig. 1. Rendering of a modern indoor environment. Created with the rendering engine **pbrt** [7]. Image credit: [8]

CCS Concepts: • **Computing methodologies** → **Rendering**; **Ray tracing**; **Reflectance modeling**.

Additional Key Words and Phrases: physically based rendering, path tracing, photorealistic rendering, global illumination

ACM Reference Format:
Sean Siddens. 2023. Computing Photorealistic Images: An Overview of Monte Carlo Global Illumination Algorithms. 1, 1 (September 2023), 19 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Rendering, in the context of computer graphics, is the process of synthesizing a 2D image from a description of a 3D virtual scene. If an image is produced that is indistinguishable from a photo taken with a *real* camera, then it is called *photorealistic rendering*.

In order to produce such an image, the "underlying physical processes regarding materials and the behavior of light" must be "precisely modeled and simulated" at a signficicant level of detail[2]. While the nature of light and how it interacts with objects has long been understood in the field of

Author's address: Sean Siddens, ssidden1@ucsc.edu, University of California, Santa Cruz.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2023 Association for Computing Machinery.
XXXX-XXXX/2023/9-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

physics, actually simulating this process accurately and efficiently on a computer has been a major focus of research in the computer graphics field for decades.

Despite the extreme challenge, photorealistic rendering has remained such an active research topic in computer graphics because it is an extremely desirable effect for a wide variety of applications. In the television and film industry, for example, it is utilized for the production of animated features. Even if the style of the film isn't strictly "photorealistic", oftentimes physically plausible lighting effects are desirable to make the film more immersive and visually impressive. Furthermore, an accurate lighting model is necessary for the visual effects industry in order to convincingly combine real and virtual elements into a single shot. Another important application of photorealistic rendering is in architecture, where indoor and outdoor lighting can be fully simulated under various atmospheric conditions and times of day, allowing architects to better design before actually constructing the building [2].

All of these applications are non-interactive since the actual computational process of rendering oftentimes takes minutes to hours to fully complete. This is due to the fact that the algorithms used in photorealistic rendering are extremely complex and computationally expensive. However, recent advances in hardware and algorithmic techniques have made it possible to achieve near photorealistic results in real-time contexts such as video games. Achieving photorealistic image synthesis in real-time speeds is a major goal of computer graphics research, since it ushers in the possibility of creating immersive virtual realities nearly indistinguishable from real life.

This paper will explore a class of photorealistic rendering techniques known as *Monte Carlo path tracing* algorithms. There are other methods which can achieve similar results, but path tracing has proven to be especially effective and efficient, especially as hardware has improved over time. Furthermore, there has been lots of research, recently and over the past few decades, into these algorithms in order to make them practically usable. There are other benefits to the path tracing algorithms which will be further discussed, but overall they are proving to be the future of photorealistic rendering in the film and video game industries [1, 3].

2 ILLUMINATION

2.1 Models of Light

There are multiple ways to model the behavior of light in computer simulations. Light is dual by it's very nature, existing simultaneously as both a wave and a stream of particles. Certain phenomena arise by assuming light adheres to one of these models. For example, the diffraction and interference effects can be explained by assuming light is a wave, and the photoelectric effect arises from assuming light is a stream of particles [2].

However, simulating a model of light which attempts to capture *every* possible physical phenomenon is unnecessary and extremely inefficient. The most common model used in computer graphics is called the *geometric* or *ray* model of light [4]. This model can be thought of as a "high level" model since it abstracts away many of the low level details of light and simplifies many assumptions. For example, the model views light as a ray, only consisting of an origin and a direction. Thus, light interacting with a surface is simply viewed as a ray intersecting some other geometric object. Additionally, it's assumed that light travels at an infinite speed and is not influenced by external effects such as gravity and magnetic fields. While the geometric model of light disallows the simulation of certain phenomenon such as diffraction and interference, it is often good enough for the purposes of computer graphics and photorealistic rendering.

Another assumption that is sometimes made is that the geometric light ray carries some particular color depending on what objects it has interacted with. In reality, light has a particular wavelength that materials absorb, thus what remains is perceived as color. This paper will make the simplifying

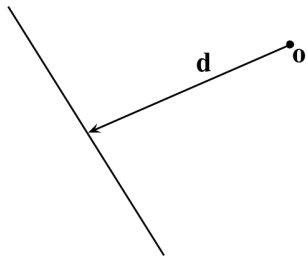


Fig. 2. Illustration of the geometric model of light. Typically, **o** and **d** are three-component vectors representing the origin and direction of the ray in three dimensional space.

assumption, with materials having a single color and rays of light transmitting this color after interacting with the material.

2.2 Reflection Models

Before light reaches a camera sensor or the eye, it is first emitted from a light source and then takes some arbitrary path throughout the scene. This "path" is governed by how light reflects with surfaces. These reflectance properties are what gives different materials their different appearances [2].

These properties can be described by a function called a *bidirectional reflectance distribution function* (BRDF). This function assumes light hits a surface at some point x from some direction Ψ and is reflected in some other direction Θ.

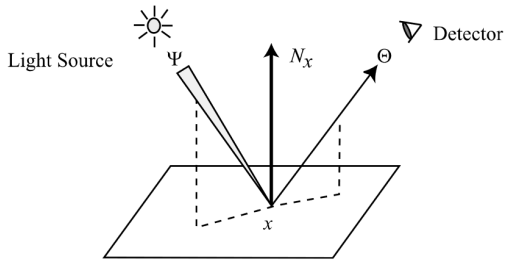


Fig. 3. Visualization of the bidirectional reflectance distribution function. Figure credit: [2].

Given the intersection point, incident, and exitant directions, the function returns the ratio of light reflected to the outgoing direction from the incoming direction. A material which reflects light equally in all directions is known as a *diffuse* or *Lambertian* material. This material will look the same when viewed from all directions, irrespective of where the light is shining from. Rough surfaces such as dirt, chalk, or unfinished wood will exhibit this property. A material which reflects light in a single direction (such as a mirror) is known as a *specular* surface. Figure 4 visually compares the different types of BRDFs and demonstrates how they ultimately describe the ratio of outgoing light to incoming light for any given direction. In reality, materials are *glossy*, meaning they exhibit a complex combination of diffuse and specular properties. Because of this, physically accurate BRDFs are often extremely complicated and cannot be described by simple analytical formulae [2].

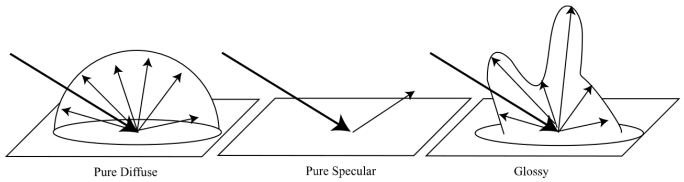


Fig. 4. Visual comparison of different BRDFs. Pure diffuse/Lambertian and pure specular materials are perfect reflectors and don't exist in the real world. A glossy BRDF, such as the one visualized above, more accureately mdoels the behavior of real life materials. Image credit: [2]

There have existed many frameworks which use these reflectance models to render images. Some of the most popular, such as the Phong or Blinn-Phong model, do not attempt to be physically accurate, but instead attempt to approximate the phenomenological appearance of materials [10]. This disregard for physical accuracy also makes these models extremely efficient, and they are often still used today in some capacity for video games. However, non-physically accurate frameworks fail to capture many of the lighting effects which give an image a photorealistic appearance.

2.3 The Rendering Equation

In order to unify all of the existing rendering frameworks into a single model, the *rendering equation* was introduced in 1986 [5]. This equation is a single mathematical formula which entirely describes the transport of light throughout a scene:

$$L(x \rightarrow \omega_o) = L_e(x \rightarrow \omega_o) + \int_{\Omega_x} f(x, \omega_i \rightarrow \omega_o) L_i(x \leftarrow \omega_i)(\omega_i \cdot n) d\omega_i \tag{1}$$

Essentially, the equation states that the total amount of light that is transported from a point x in a direction ω_o is the sum of the emitted light and the total amount of light which is reflected to x from all other directions on the hemisphere centered above x [5].

The L_e term is the *emmittance term* and accounts for how much light is emitted from the point. If the equation is being evaluated on a point on a light source, such as a desk lamp or the Sun, this term would be non-zero. For any other point in the scene, however, this term should be zero. The other half of the equation integrates over the hemisphere Ω_x, accounting for all of the light contributing to the point x from all other directions ω_i on the hemisphere. Figure 5 visualizes this integral and how it relates to the incoming and outgoing directions of light.

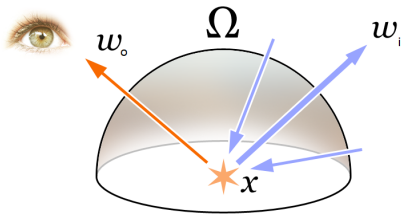


Fig. 5. The rendering equation integrates over all incoming light arriving to the point from other points in the scene. Image credit: [11]

The first term f inside the integrand is the BRDF mentioned earlier, which determines the ratio of light which contributes to the point from the incoming direction that is reflected to the outgoing direction.

The next term is actually a *recursive call to the rendering equation itself*: how much light is coming from *different* directions in the scene needs to be computed. This recursive call makes the rendering equation extremely challenging to solve. Analytically computing it is infeasible, and using other approximations can be extremely challenging. Fundamentally, the process of photorealistic rendering is about computing this integral, and the goal of the algorithms which will be addressed in this paper is to find ways to *estimate* this integral quickly and efficiently.

The final term in the integrand is what is known as the *geometric term*. For two normalized vectors a and b (it is assumed that every vector quantity is normalized), their dot product gives the cosine of the angle between them:

$$a \cdot b = cos\theta$$

The geometric term is the dot product between the incoming light direction and the surface normal. This term accounts for how the alignment of these two vectors effects how much light is actually deposited on the surface point from the incoming direction. As the incoming direction approaches being perpendicular to the surface normal (and therefore *parallel* to the surface), the term approaches zero. At glancing angles, light is spread over a larger area, so the actual contribution lessens. If the light is parallel to the surface, no light is deposited, and the term should be zero. If the incoming direction is directly over the surface, then the angle between the surface normal and the direction will be zero, thus the term would be one. Intuitively, this term is the reason why midday is the brightest time of day, while sunset and sunrise are the darkest: light at glancing angles to a surface contributes far less than light at a direct angle.

One may notice that the model of how paths of light are traced with the rendering equation is opposite of how they flow in reality: light is followed starting from the camera to the light source. This model greatly increases the efficiency of the algorithms used to approximate the rendering equation. What makes this optimization possible is *Helmholtz' law of reciprocity*, stating that the journey a ray of light and it's *reverse* takes is the same [4]. What this means is that the value of the rendering equation is the same whether light is traced starting from the light source or from the camera.

2.4 Global Illumination

The rendering equation encodes all of the behavior of light transport needed to capture many complex lighting phenomenon. When evaluating the integral of the equation, if the incoming light direction comes directly from a light source, this is known as *direct lighting*. On the other hand, if this direction comes from another point in the scene which isn't a light source (not intrinsically emitting any light), this light contribution is called *indirect lighting*. The combination of these two phenomenon captures the behavior of all types of light scattering throughout a scene, and is called *global illumination* [4].

Figure 6 captures the different lighting behaviors. Indirect lighting comes from the fact that light often bounces off multiple surfaces before it hits the final surface. Without indirect lighting, as figure 6(b) illustrates, points not in direct line of sight with a light source will appear completely black. Furthermore, in figure 6(d) where all lighting effects have been combined, it can be see that the white walls don't appear 100% white. This is again due to indirect illumination, where light bouncing off one of the colored walls will hit the white wall, contributing some of it's color to the wall's appearance.

In figure 6(d), one can also see bright highlights inside the shadow of the glass teapot on the right. These bright spots are known as *caustics*, and are a result of light travelling from a light source, refracting through a glass object or reflecting off a mirror-like object one or more times,

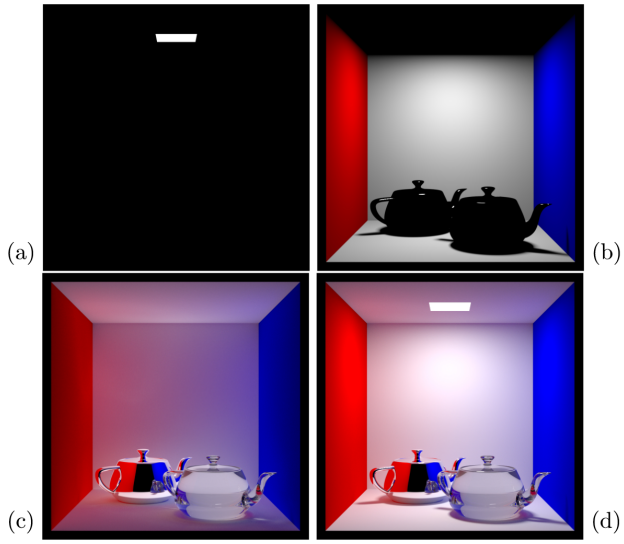


Fig. 6. The different components of lighting: (a) the light source, (b) direct lighting, (c) indirect lighting, (d) all combined (global illumination). Image credit: [1]

and then hitting a diffuse surface [4]. Essentially, multiple rays of light are concentrated to a single point, resulting in brighter spots in the image.

These complex lighting behaviors are why the rendering equation is so powerful. Under this framework, one can completely capture these global illumination effects. However, as will be explored further, simulating these effects is tremendously hard. Many real-time video games more closely resemble figure 6(b), since capturing direct lighting is far easier than indirect. Oftentimes in video games or even movies these global illumination effects were "faked" using textures or colored light sources, giving the illusion of indirect lighting without actually having to simulate it [4].

3 MONTE CARLO METHODS

In order to construct a computer simulation which evaluates the rendering equation, one needs a strategy for numerical integration. The most well known method for this is the *Riemann sum*, where the region to be integrated is partitioned into many regular shapes, such as rectangles, and then their aggregate area will be similar to the actual value of the integral [13].

The Riemann sum to evaluate the integral of the function f is defined as:

$$S = \sum_{i=1}^{n} f(x_i)\Delta x_i$$

where n is the number of partitions the region has been divided into, and Δx_i is the width of one partition. The function is sampled at some fixed location within the sub-interval defined by each partition.

The issue with this technique is that if the function of interest is a two dimensional function (such as the surface integral in the rendering equation), n^2 partitions need to be summed to get the same accuracy. The number of samples needed for the sum will grow exponentially with each dimensional increase. This is known as the *curse of dimensionality* [13]. For the problem of

solving the rendering equation, a numerical integration technique which has the same performance *regardless of dimensionality* is needed.

3.1 Monte Carlo Integration

The solution to this problem is through a class of mathematical techniques called *Monte Carlo methods*. These are algorithms which use a random process to generate a solution. These techniques are heavily reliant on probability theory, so a brief review is in order.

For a continuous random variable x, the *probability density function* (PDF) $p(x)$ defines the probability the variable will take on some given value [2]. The *expected value* of a random variable can be thought of as the average value the variable would take on over some probability distribution, and is defined as:

$$E[x] = \int_{-\infty}^{\infty} xp(x)dx.$$

The *variance* is the average of how far the random variable deviates from its expected value, and is defined as [2]:

$$\sigma^2 = E[(x - E[x^2])] = E[x^2] - E[x]^2 \tag{2}$$

The variance is extremely important for measuring the error of these algorithms, and can be a useful metric for comparing their relative performance and efficiency.

Let I be the integral of interest, defined over $[a, b]$:

$$I = \int_a^b f(x)dx. \tag{3}$$

The *Monte Carlo estimator* for the integral is defined as

$$\langle I \rangle = \frac{1}{N} \sum_{i=1}^{N} \frac{f(x_i)}{p(x_i)} \tag{4}$$

where each sample x_i is taken from the PDF $p(x)$. This estimator is considered *unbiased*, since the expected value of the estimator is the actual value of the integral [2]. This means that an extremely accurate estimate of the integral can be acquired as long as a sufficient number of samples are taken. Furthermore, this strategy does not suffer from the curse of dimensionality. Regardless of whether the function is one or two dimensional, the same number of samples are required to converge to a good estimate.

The variance of the estimator is

$$\sigma^2 = \frac{1}{N} \int (\frac{f(x)}{p(x)} - I)^2 p(x)dx.$$

This demonstrates the one weakness of Monte Carlo integration: the variance decreases only linearly with N. This means that fully removing the error from an estimate can take an extremely large number of samples.

The way Monte Carlo integration is used in the context of photorealistic rendering is for giving an estimate of the rendering equation. Exactly how this is done will be explained in a later section, but essentially the samples of the estimator will be taken over the hemisphere around the point to be shaded, allowing for the estimate of the value of the rendering equation.

4 MONTE CARLO PATH TRACING

Monte Carlo integration is a fundamental component of the *path tracing* algorithm. However, the fundamentals of how images can be generated with *rays* must first be explained.

4.1 Ray Casting

Imagine looking out through a window inside a comfy isolated cabin, gazing at an idyllic mountain vista. If one were tasked with perfectly replicating this view onto a canvas, how might the problem be optimally approached by an unskilled painter? One strategy would be to hang a square-latticed net over the window. For each square in the net, one simply has to copy the color seen through the square onto corresponding location on the canvas. The smaller the holes of the net, the more accurate the painting will appear.

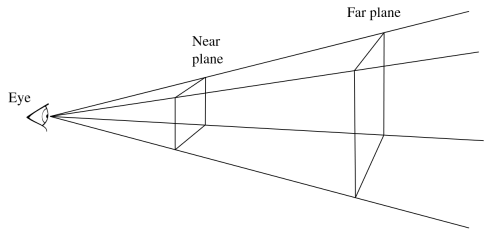


Fig. 7. Visualization of how the *painter's algorithm* (also known as *ray casting*) can be used to render an image. The near plane is the image or "window" that is being looked through, and the far plane would be the limits of the simulation (rays cannot be traced forever). Image credit: [9].

This is fundamentally how *ray casting* is used to render images: rays are *cast* through each pixel in the image and out into the scene and whatever color is "seen" by that ray is what is used to color the final pixel. Increasing the resolution of the pixels, thus shooting more rays, produces a more accurate photo.

This algorithm is powerful since it correctly captures *perspective projection* and solves the *hidden surface problem*. Perspective projection is responsible for making objects farther away look smaller and closer objects look bigger. This is essential in photorealistic rendering because it replicates how humans perceive the world. The hidden surface problem is the issue of figuring out whether a given point on an object is visible to the camera. This can be extremely challenging to solve when not using ray casting, since one would have to determine visibility for every point on every object in the scene. By casting rays from the eye, only surfaces actually visible will ever need to be considered.

4.1.1 Ray-Scene Intersections. A key component of the ray casting algorithm is determining the closest point of intersection a ray had with the scene. With the geometric model of light, a ray is represented with an origin $\boldsymbol{o}$ and a direction $\boldsymbol{d}$. Given some positive value t, one can parameterize the ray to extend it to any given point along the ray [9]:

$$r(t) = \boldsymbol{o} + \boldsymbol{d}t$$

Using this parameterization, one can solve for the closest intersection by simply finding the point of intersection with an object in the scene such that the t value is the lowest.

It is trivial to compute this value for any geometric surface which can be described by an explicit function, such as a triangle or a sphere. In computer graphics, it is most common to represent a

scene using triangles, so much research has gone into developing fast and efficient ray-triangle intersection routines. One of the most commonly used algorithms is the Möller-Trumbore algorithm [6], which is widely used due to it's speed, efficiency, and simple implementation.

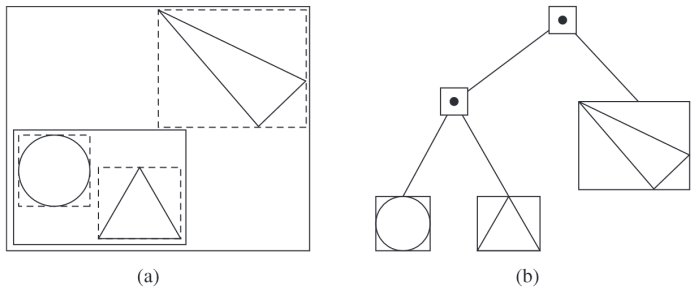


Fig. 8. Diagram of the relationship between a simple scene (a) and it's BVH representation (b). Figure credit: [9]

In a ray casting program, the intersection routine will oftentimes still be the most expensive part, since a scene can be comprised of millions of triangles, and each ray needs to test it's intersection against every single triangle. One essential optimization which can be done is the use of a spatial acceleration structure such as a *bounding volume hierarchy* (BVH), which allows rays to not have to consider every single triangle while intersection testing. A BVH constructs a tree of nested bounding boxes around the triangles in the scene. Then, an intersection test is done by traversing the tree, carrying out an intersection against the box at each node, until it eventually reaches a leaf node containing the actual triangle. The runtime of the intersection routine goes from being quadratic with respect to the number of triangles to sub-linear. Millions of triangles are culled from ever having to be intersected against, which greatly speeds up a ray casting program.

4.2 Ray Tracing

The ray casting algorithm discussed thus far does not describe how exactly to choose what color to put for each pixel in the image. It only considers where rays intersect in the scene.

In 1980, Arthur Whitted introduced the *ray tracing* algorithm as a means of both rendering the geometry of a scene, but also modelling it's *illumination* [12]. The key insight of this paper was that light could be simulated by tracing it from the camera instead of the light source. This meant that the ray casting algorithm is used to send out what are known as *view rays*. Then, for each intersection in the scene by a view ray, a computation of how to shade that point can be done by *shooting more rays*. One way to do this is by tracing a ray to the light sources in the scene. One can determine the amount of light which is being contributed from that light source based on the distance from and angle between the surface and the light source. Whether the point being shaded is actually visible to the light source can also be determined by shooting additional rays If the point isn't visible to the light source, then the point is in shadow.

Another way this technique can be utilized is for rendering mirror-like objects. When a ray intersects this material, a new ray is traced such that it's direction is a reflection of that incoming ray. This simulates the behavior of light rays with specular objects. Furthermore, refraction by glass-like materials can also be simulated in a similar manner by generating a ray based on Snell's law of refraction. Figure 10 demonstrates this, where the checkerbox plane can be seen in the reflection of the left sphere and refracted through the right sphere. This also means that complex

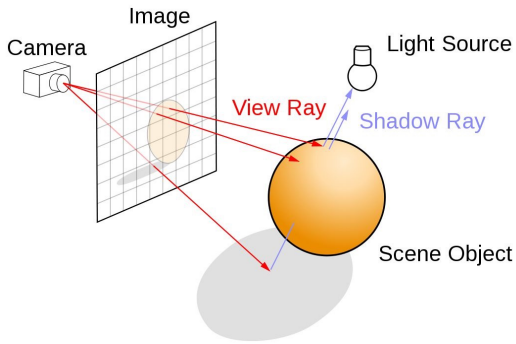


Fig. 9. Diagram of how the ray tracing algorithm is used to shade points. Additional rays can be traced to determine if the point is in shadow (shadow ray is obstructed) and also determine the distance from and angle to the light source to determine light contribution amount. Image credit: [14]

lighting behavior such as interreflection between multiple mirrored objects can be easily captured by simply tracing a ray many times throughout a scene.

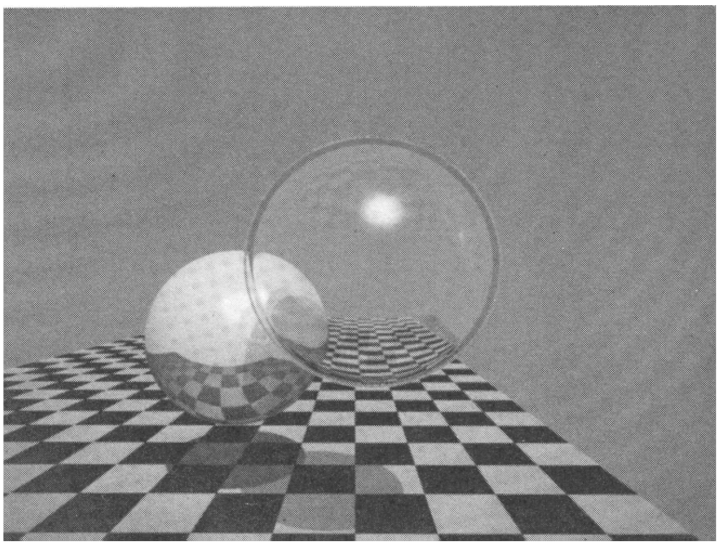
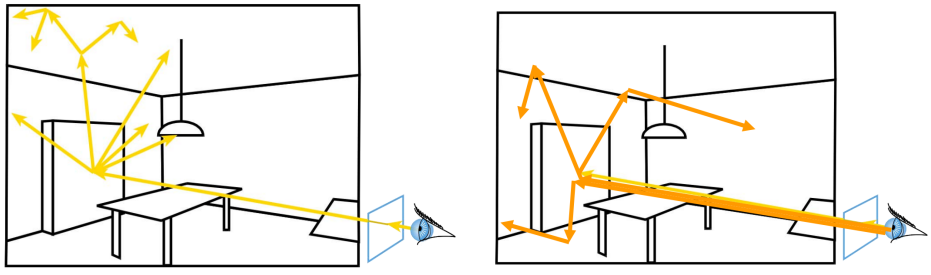


Fig. 10. Rendering from Whitted's seminal 1980 paper demonstrating the different complex illumination effects which can be captured by the ray tracing algorithm [12].

The limitation of ray tracing is that it makes no attempt to compute the rendering equation, so it will not capture physically based global illumination behaviors. Furthermore, the ray tracing process creates a tree of child rays. This exponential branching factor can make it impossible to trace rays any significant depth.

This simple ray tracing algorithm is now referred to as *Whitted-style ray tracing* or *classic ray tracing* [2]. In modern use, the algorithm has since been built upon in order to capture more complex lighting effects. One possible application of the ray tracing technique is to use it in order to estimate the rendering equation. By doing this, it is possible to converge to an actual evaluation of the



(a) MC ray tracing (b) MC path tracing

Fig. 11. Comparison of the Monte Carlo ray tracing (a) and Monte Carlo path tracing (b) algorithms. MC ray tracing results in many branching recursive calls due to the ray tree. MC path tracing only takes one sample per hemisphere, but many paths need to be traced per pixel. Image credit: [15]

rendering equation (given enough samples). For every surface intersection, the integral can be estimated by shooting N child rays out into scene, distributed equally over the hemisphere around that point. This then forms a Monte Carlo integrator for that integral. These child rays are shot out into the scene and will recursively evalutate the rendering equation for wherever they intersect in the scene. The recursion ends either when a specified depth limit has been reached or the ray reaches a light source. At that point, the emittance term of the equation would be non-zero, and the integral at that point wouldn't have to be evaluated. This algorithm is known as *Monte Carlo ray tracing*.

4.3 Path Tracing

Monte Carlo ray tracing will an unbiased estimate of the rendering equation, and will converge to the solution given enough samples. However, the exponential branching factor due to taking multiple samples of the integral still needs to be addressed. One solution is to simply take *one* sample for each estimate of the integral. This strategy will simulate a single path of light through the scene for each initial view ray that is shot. This algorithm is what is known as *Monte Carlo path tracing*.

A basic outline of the algorithm is as follows [13]:

(1) For each pixel, shoot a ray starting at the eye that extends through the pixel and into the scene.
(2) Compute ray-scene intersection information for the hit surface point.
(3) If surface point emits light ($L_e > 0$), return this value. If not, then the reflected energy from other points in the scene to this point needs to be estimated, which is done by taking a *single* sample.
(4) Sample is a random direction ω_i on the hemisphere, selected according to some PDF. A ray is shot in that direction in order to sample incoming light coming from somewhere else in the scene, thus extending the path by one segment.
(5) Algorithm continues on step 2, terminating when the path reaches a light source or leaves the scene.

In isolation, each path will be a poor approximation of the rendering equation, but if enough paths are traced, the aggregate of these paths will converge to the actual value of the rendering equation. One way to reason about how the algorithm works is that it essentially simulates a subset of all possible paths photons take from the light source and to the eye/sensor. Images are simply

the results of photons being emitted from some light source, bouncing around, then eventually making it to the camera. The algorithm simply simulates a subset of those paths. As more paths are traced, the image will get better. This is known as a *consistent* estimator: as more samples are taken, the error will go to zero, and the estimate will converge to the actual value of the function [2].

4.3.1 Ray Termination. One may run into issues with the algorithm described above in scenes where light paths could potentially be extremely long due to light bouncing many times before eventually making it to a light source or leaving the scene. Because of this, most implementations use some method to terminate the path tracing procedure once a certain condition has been met. The simplest solution is to cut off all paths of a certain length N. If the path being traced reaches this length, the recursion is stopped and the path won't have any contribution to the image.

The issue with this strategy is that some of the long paths that are being terminated can potentially have a large contribution to the image. The optimal technique used (known as *Russian roulette*) keeps path lengths from ballooning to infinity but still allows the potential of exploring these long paths [2]. Before each step of the path tracing procedure, a probability $\alpha = 1 - P$ of terminating the path and stopping the recursion is assigned. α is known as the *absorption probability*. In order to keep the estimator unbiased, the sample must be weighted by $1/P$. If α is large, then the path lengths will be cut short more often, and the variance will be higher, but if it's too small, path lengths may become too long. Russian roulette won't decrease variance (in fact, it often increases it), but it can improve *efficiency* by skipping paths which are likely to not make a large contribution [9]. The absorption probability can be chosen dynamically based on the material, such as making it higher for dark materials. This can reduce the likelihood of increasing variance while still improving overall efficiency.

5 REDUCING VARIANCE

The simplest version of the MC path tracing algorithm will produce accurate images, but the estimator is still extremely high in variance. This means than an extremely large number of samples is needed to converge to a good looking image absent of noise. Some scenes, such as very dark scenes with few light sources, may practically be impossible to render fully without noise, since doing so may require tens of millions of paths to be traced per pixel. If the variance can be reduced, then a better looking image can be computed with far less samples (and thus in far less time). Because of this, much of computer graphics research in this field has been dedicated to figuring out techniques which reduce the variance of the Monte Carlo estimator used for the rendering equation.

5.1 Importance Sampling

The variance of the estimator described in equation 4 can be reduced by taking samples from a probability distribution $p(x)$ that is similar to the function $f(x)$ in the integrand [9]. The idea behind this is that it is better to choose more samples which will have a higher contribution to the final value of the equation. For example, if the probability distribution sampled a higher proportion of directions at glancing angles from the surface, time would be wasted computing light paths which won't end up contributing anything, since the geometric term will cause the whole integrand to go to near zero.

This can be proven with the definition of variance described in equation 2:

$$Var[F] = E[F^2] - E[F]^2$$

where F is the estimator. $E[F]$ is just the value of the integral being estimated, $E[F^2]$ needs to be minimized. Recall that $F = f(x)/p(x)$, so then it is desirable to have $p(x)$ large where $f(x)$ is also large, keeping the term small [17].

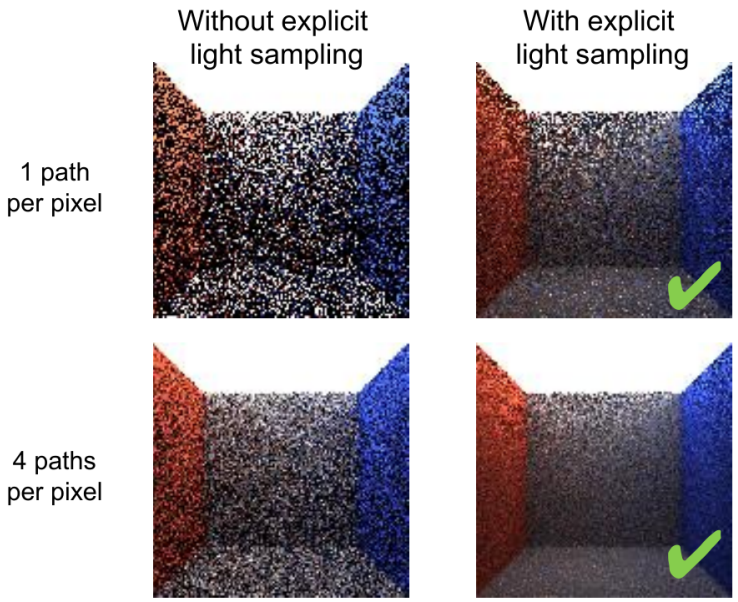


Fig. 12. Demonstration of the benefit of importance sampling. Image credit: [15]

In practice, this means having a sampling distribution which closely matches the BRDF. For example, the ideal PDF for the pure diffuse BRDF illustrated in figure 4 would be one which samples directions equally on the hemisphere. An ideal PDF for the specular surface would be one which directed samples around the point of reflection, since that's where most of the actual light contribution will be coming from.

Another technique, known as *light sampling*, samples from directions which go towards the light source. This is because the light energy from these directions will dominate the total contribution to the point visible to the light source, so it's better to direct more samples in that direction instead of generating them randomly. As can be seen in figure 12, this technique can have significant performance improvements, especially in a dark scene with few light sources.

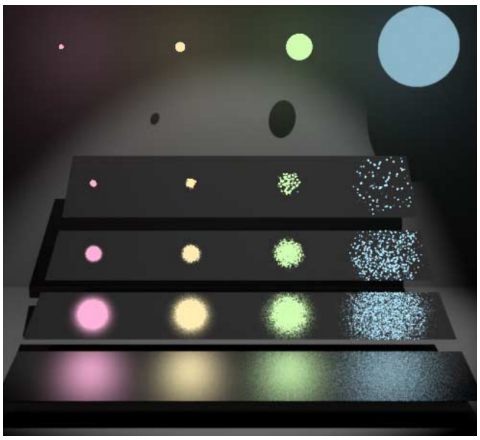
5.2 Multiple Importance Sampling

The issue with choosing a PDF which matches the function being integrated is that, for the case of rendering, the function is very complicated. For one thing, the integrand in the rendering equation is a combination of many terms. Additionally, the function varies widely depending where in the scene and in what direction it's being evaluated. Because of this, sticking to a single sampling strategy, such as BRDF or light sampling, won't be suitable for the entire scene. Figure 13 demonstrates this idea: certain sampling strategies have strengths and weaknesses in different parts of the scene. Again, this is simply due to the fact that the rendering equation is composed of multiple terms and highly varies depending on where in the scene the sample is being sent from.

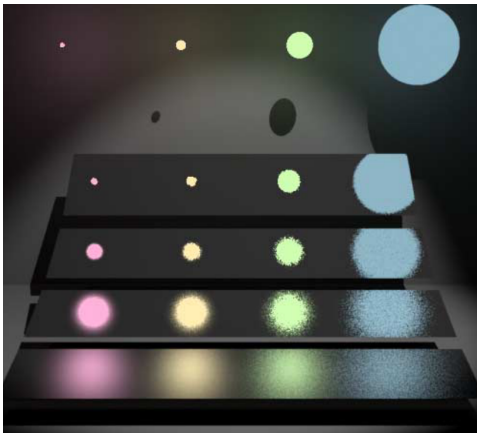
The optimal solution is then to use multiple sampling strategies, combining them into a single strategy which preserves each of their strengths. This known as *multiple importance sampling*,

and is achieved through a *weighted combination* of the different strategies. A simple averaging of samples taken from different distributions is insufficient due to the fact that variance is additive. Once the variance is there, averaging the different sampling strategies will not get rid of it. The key idea is to *weight* the sample from each distribution in a way which reduces the most variance [16].

(a) BRDF sampling



(b) Light sampling



(c) Combination of sampling strategies using MIS.

Fig. 13. Multiple importance sampling combines multiple sampling strategies to capture strengths of each. Image(s) credit: [16].

The heuristic developed which was created to find this optimal weighting is called the *balance heuristic*. If the value sampled from a distribution p at a point x is low, but the distribution is a good match for the function f, then the value at $f(x)$ should also be low. However, if the other probability distributions are high at that sampled point, the weights need to be created such that the sample from p is weighted much higher than the other samples. This process can be done adaptively across the domain of the function being estimated (e.g. at any point in the scene), thus leveraging the use of multiple sampling strategies while preserving the strength of each one. Figure

13(c) shows that combining the two sampling strategies in this manner results in a far less noisy image compared to if only one of the strategies were used for rendering the entire scene.

5.3 Bidirectional Path Tracing

One method for variance reduction which was first introduced in 1995 was *bidirectional path tracing* [17]. This method builds upon the path tracing algorithm by forming paths which start from both the eye and the light source simultaneously.

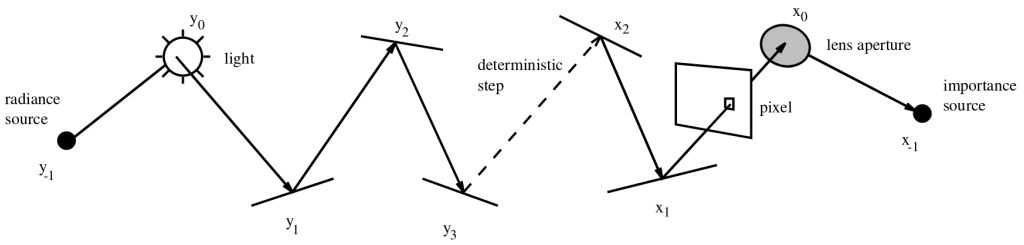


Fig. 14. Light paths are constructed from the eye and the light source. Figure credit: [17]

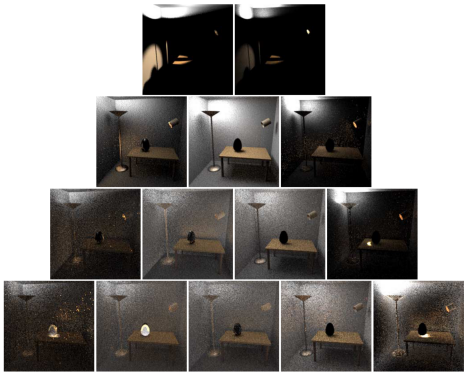
Figure 14 illustrates the algorithm: a path can be constructed simultaneously from the light and the eye. Each step of these paths are random due to the nature of Monte Carlo integration, but the connection of the two paths is deterministic. The choice of where to place the partition between the two paths is explicit: if the total path length is k, then there's k choices to partition the entire path into m eye steps and n light steps (where $m + n + 1 = k$) [17].

Each choice of how to partition the path results in an estimator with a different variance (see figure 15(a)). The algorithm developed calculates the optimal combination of these estimators which results in the lowest overall variance. This technique essentially uses multiple importance sampling to optimally weight the different path contributions. Figure 15 shows how this technique successfully results in rendered images which have significantly less noise than standard path traced images, but is achieved with the same amount of work.

Bidirectional path tracers perform extremely well for certain scenes which exhibit tricky lighting behavior. For example, consider the scene in figure 16, where the scene is being illuminated solely from indirect illumination coming from a single light source enclosed in a wall sconce. If rays are only traced starting from the camera, "most of the paths will have no contribution, while a few of them — the ones that happen to hit the small region on the ceiling — will have a large contribution " [9]. Tracing a path from the camera which happens to make it back to the light source is extremely unlikely. The result of this is an extremely noisy final image due to this high variance. However, if rays are traced using the bidirectional technique, more paths will be traced which directly connect to the light source, resulting in a far less noisy image for the same number of samples.

6 HARDWARE ACCELERATION

Even with all of the techniques to speed up Monte Carlo path tracing discussed thus far, practically rendering an entire feature length film is extremely challenging. In a 90-minute movie, there are about 130,000 frames [1]. Each frame contains at least two million pixels (at least four million for IMAX). Therefore, at least 260 *billion* pixels need to be computed for the final render of an animated film. No matter how much algorithmic speedup can be gained, performing each one of these computation tasks one at a time is a fool's errand.



(a) Each choice of partitioning the light and eye paths can capture different aspects of the scene lighting and results in a unique estimator.

(b) The bidirectional techniques can all be combined into a single image (left) which results in a better result than just through standard path tracing (right) in the same amount of work.

Fig. 15. Demonstration of how the individual bidirectional techniques can capture different aspects of the image (a) and be combined into a final image (b). Image credit: [16]

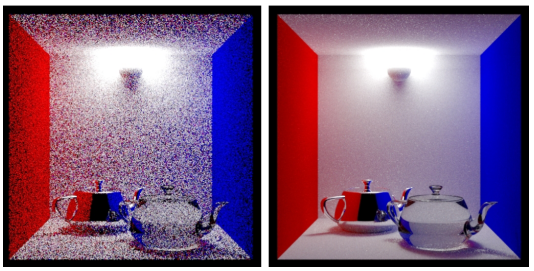


Fig. 16. Box illuminated by a light source within a wall sconce. Left is rendered by a traditional unidirectional path tracer, while the right is from a bidirectional path tracer. Bidirectional tracing results in far less noise. Image credit: [1]

This is where *hardware acceleration* comes in. The Monte Carlo path tracing algorithm thus described is *embarrassingly parallel*, meaning that the work can be evenly split up into multiple independent tasks without needing to worry about synchronization or data races. Each path traced is essentially it's own isolated routine which only requires a description of the scene data, not needing to modify any global state. The only synchronization involved in the entire algorithm would be combining all of the paths into a single image. Because of this, MC path tracing is primed for the leveraging of hardware-accelerated optimizations which execute the algorithm in parallel.

6.1 SIMD Ray Tracing

In 1999, Intel introduced *single instruction multiple data* (SIMD) instructions to their CPUs [1]. These instruction essentially allow for the execution of multiple (often 4 to 8) computations to be carried out at the same time. This is achieved using specialized hardware, meaning that using these SIMD instructions where needed can result in far greater speedups than if a single instruction were just repeated.

The obvious application of these instructions is for the tracing of multiple rays at a time. This can be used to trace coherent rays which are near each other by testing their intersections against a single triangle. Or, a single ray intersection can be tested against multiple triangles simultaneously. Additionally, SIMD instruction can be used to used to quickly traverse the BVH by intersection testing against multiple bounding boxes at the same time [1].

6.2 Multithreading

As mentioned earlier, the nature of the path tracing algorithm means that rendering software can be easily refactored to leverage parallelism. Modern CPUs contain many cores which can execute code in parallel. Programmers can utilize these cores by splitting up code to be executed by multiple *threads*. The most straightforward and effective manner of dividing up path tracing work on multi-core CPUs is to simply have each thread render a chunk of the image.

Scaling issues with this technique can occur due to reality that all of the data needed for rendering (scene and material data) cannot be present in the cache at the same time. Memory access is oftentimes incoherent, meaning that L1 cache data will be frequently evicted [1]. This issue is only further exasperated with the introduction of additional cores.

Regardless of this practical limitation, designing a renderer with hardware in mind is the only way to achieve maximum performance. Intel has created their own ray tracing framework known as Embree which is designed to best utilize their x86 CPU capabilities to accelerate ray intersection tests and BVH traversal [18]. They provide an API which can be easily introduced to existing renderers without having to do a full refactor. By having a framework which explicitly targets a specific hardware architecture, they can achieve significant performance gains over a renderer which doesn't utilize any hardware acceleration.

6.3 GPU Ray Tracing

GPUs are specialized hardware which are best suited for massively parallel computational tasks. They were developed for the use in computer graphics rendering with the triangle rasterization algorithm. Over time, their architecture and programming model has slowly switched to focusing on general computational tasks instead of fixed-function graphics rendering. This can be seen in the rise of GPU usage in non-graphics domains such as cryptocurrency mining and the training of AI models.

This general compute capability can be leveraged for path tracing. The simplest way to do this is to write the entire renderer inside a *pixel shader*, which is a GPU program which runs in parallel for every pixel in an image being rendered. The pixel shader would be in charge of determining the color of the final image using the Monte Carlo path tracing algorithm discussed thus far. This will be referred to as the *megakernel* approach, since it involves fitting the entire path tracing algorithm into a single GPU program (kernel).

The issue with this strategy is that a naive direct port of the CPU algorithm will not fully utilize the full hardware capabilities of the GPU. Certain pixels in the image (such as those which have simple materials or contain simple geometry) will execute much faster than other pixels, which results in severe hardware under-utilization. This is only further exacerbated by control flow divergence due to branch statements in the algorithm. Every core on the GPU executes the same instruction in lock-step, *regardless of what branch is taken*. Every control flow path is executed by every single core, resulting in an immense amount of needless computation [19].

Furthermore, GPU memory has much higher latency than CPU memory. To address this, GPUs have many threads (far more than the actual number of cores) execute concurrently so that threads can be switched out whenever they wait for a memory fetch. The preemption and scheduling overhead is negligible compared to the speedup gained from hiding the memory latency issues.

However, the megakernel approach restricts the number of threads which can be ran concurrently due to the fact that each thread will require far more registers. Every core shares a single register file, so fewer threads overall will be able to be ran concurrently without register spilling [19].

The *wavefront path tracing* algorithm was developed to address these issues [19]. This algorithm splits up the megakernel strategy into four separate smaller programs. These four smaller programs represent phases of the path tracing algorithm, and their work is comprised of rays instead of pixels. This strategy better avoids the issues of workload imbalance, control flow divergence, and register spilling, thus better utilizing the full capabilities of modern GPU hardware.

6.3.1 Denoising. One of the properties of a Monte Carlo estimator is that increasing the number of samples has diminishing returns. This means that when rendering an image, about 95% of the noise can be removed fairly quickly, but that last 5% can be extremely challenging to fully remove.

Instead of tracing millions of more paths to resolve a tiny portion of the image, it is far more worth it to use a post-process effect known as *denoising* instead. GPUs are extremely good at image processing and are always used for these algorithms. The most basic form of denoising is just a simple blur, which averages the color of each pixel with it's neighbors' colors. The issue with this is that it isn't "feature-guided". Noise from diffuse reflection can be different than noise from specular reflections. Additionally, the denoising process should preserve the detailed parts of the scene and object edges [1].

The solution is to have a denoising strategy which takes all of these features into consideration. This means that when processing each pixel, it has knowledge of "the average depth, surface normal, motion vector, surface albedo, specular and diffuse reflection, as well as variance estimates" [1].

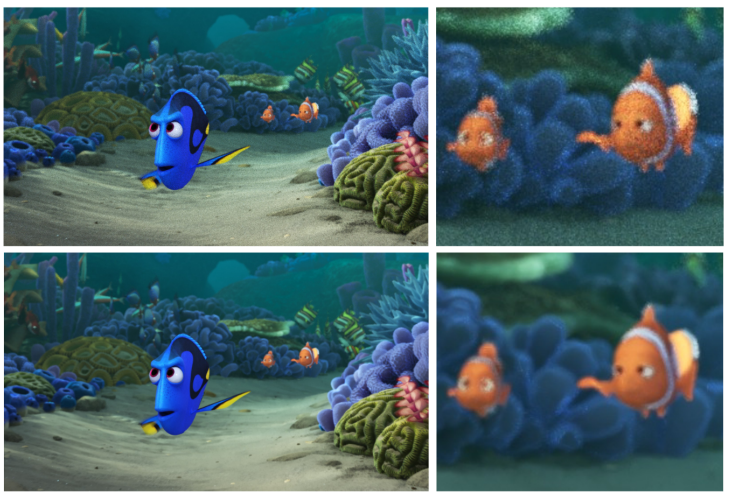


Fig. 17. Comparison of a noisy frame and a frame which has been denoised. Image credit: [1]

Another strategy is to use machine learning models for denoising. NVIDIA's OptiX™ AI-Accelerated denoiser was trained on tens of thousands of images of rendered scenes [20]. Then, it was taught remove the noise from these images to most accurately reach the original image. These neural network denoisers are proving to be extremely good at the denoising process, and allows for much less of the image to actually be computed, since the denoiser can simply reconstruct the finer details.

7 CONCLUSION

Since its inception, the field of computer graphics has strove to develop techniques which best replicate and simulate the natural world. Physically accurate light transport is the single most important aspect to a photorealistic rendering. Without it, the image is destined to remain artificial. Since Kajiya introduced the rendering equation in 1986, it has been the dream of researchers to develop a method which perfectly solves the light transport problem in the most efficient way.

However, simulating light to such a level that it can fool the human eye takes immense computational power and countless clever optimizations. The utilization of Monte Carlo methods for solving the underlying algorithm allows for an estimation to be good enough for solving the problem, but this estimate still extremely hard to compute. The source of the amazing performance improvements of Monte Carlo path tracing has been thanks to a combination of the development of new algorithmic techniques and rapidly progressing new hardware.

Research into light transport and physically based rendering is not slowing down. Although this paper mainly discussed relatively old techniques which have been around for years, there have been many modern algorithm and hardware breakthroughs in recent years. The demand for immersive media will only get stronger as the technology gets better and better. Real-time photorealistic rendering will be the technology which unlocks the feasibility of completely immersive virtual reality, and will usher in a new age of human-digital interaction.

REFERENCES

[1] Christensen, P.H. and Jarosz, W. 2016. The path to path-traced movies. Foundations and Trends® in Computer Graphics and Vision 10, 2, 103–175.

[2] Dutre Philip, Bala, K., and Bekaert, P. 2006. Advanced global illumination. CRC Press.

[3] Fascione, L., Hanika, J., Pieké, R., et al. 2018. Path tracing in production. ACM SIGGRAPH 2018 Courses.

[4] Jensen, H.W. 2010. Realistic image synthesis using photon mapping. Peters, Wellesley, MA.

[5] Kajiya, J.T. 1986. The rendering equation. Proceedings of the 13th annual conference on Computer graphics and interactive techniques - SIGGRAPH '86.

[6] Möller, T. and Trumbore, B. 1997. Fast, minimum storage ray-triangle intersection. Journal of Graphics Tools 2, 1, 21–28.

[7] Pharr, M. MMP/PBRT-V4: Source code to pbrt, the Ray Tracer described in the forthcoming 4th edition of the "physically based rendering: From theory to implementation" book. GitHub. https://github.com/mmp/pbrt-v4.

[8] Pharr, M. MMP/PBRT-V4-scenes: Example scenes for pbrt-V4. GitHub. https://github.com/mmp/pbrt-v4-scenes.

[9] Pharr, M., Jakob, W., and Humphreys, G. 2017. Physically based rendering from theory to implementation. Morgan Kaufmann Publishers, Amsterdam.

[10] Phong, B.T. 1975. Illumination for computer generated pictures. Communications of the ACM 18, 6, 311–317.

[11] Rendering equation. 2022. Wikipedia. https://en.wikipedia.org/wiki/Rendering_equation.

[12] Whitted, T. 1980. An improved illumination model for shaded display. Communications of the ACM 23, 6, 343–349.

[13] Bikker, J. Probability theory for physically based rendering. Jaccos Blog. https://jacco.ompf2.com/2019/12/11/probability-theory-for-physically-based-rendering/.

[14] Ray Tracing. 2019. NVIDIA Developer. https://developer.nvidia.com/discover/ray-tracing.

[15] Justin Solomon. 2021. Introduction to Computer Graphics (Lecture 16): Global illumination; irradiance/photon maps. YouTube. https://youtu.be/odXCvJTNn6s.

[16] Veach, E. and Guibas, L.J. 1995. Optimally combining sampling techniques for Monte Carlo rendering. Proceedings of the 22nd annual conference on Computer graphics and interactive techniques - SIGGRAPH '95.

[17] Veach, E. and Guibas, L. 1995. Bidirectional estimators for Light Transport. Photorealistic Rendering Techniques, 145–167.

[18] Wald, I., Woop, S., Benthin, C., Johnson, G.S., and Ernst, M. 2014. Embree. ACM Transactions on Graphics 33, 4, 1–8.

[19] Laine, S., Karras, T., and Aila, T. 2013. Megakernels considered harmful. Proceedings of the 5th High-Performance Graphics Conference.

[20] Chaitanya, C.R., Kaplanyan, A.S., Schied, C., et al. 2017. Interactive reconstruction of Monte Carlo image sequences using a recurrent denoising autoencoder. ACM Transactions on Graphics 36, 4, 1–12.