CSCD95 Final Report

A report on the nuts and bolts of a pathtracer

Leo (Sunpeng) Li
999093421

TABLE OF CONTENTS

1 The Rendering Equation

Let's jump straight into the rendering equation. Regardless of what rendering algorithm you use, its end goal is to compute the result of the rendering equation, as described by Kajiya[i]. Don't worry too much about what each component means, as we will go into detail later in this section.

$$L_o(p, \omega_o) = L_e(p, \omega_o) + \int_{\Omega} f_s(p, \omega_i, \omega_o)\, L_i(p, \omega_i)\, (\omega_i \cdot n)\, d\omega_i$$

Where (note that all vectors are assumed to be normalized):

- ω_i, ω_o = Direction of incoming and outgoing radiance respectively
- p = The surface point at which we are calculating the rendering equation.
- n = Surface normal at surface point p
- $L_o(p, \omega_o)$ = Radiance output at surface point p in direction ω_o
- $L_e(p, \omega_o)$ = Radiance emitted at surface point p in direction ω_o
- Ω = The hemisphere around surface point p containing all possible ω_i's
- $f_s(p, \omega_i, \omega_o)$ = The BSDF: proportion of light scattered (reflected/transmitted) from direction ω_i to ω_o at surface point p.
- $L_i(p, \omega_i)$ = Incoming radiance towards surface point p from direction ω_i
- $(\omega_i \cdot n)$ = weakening factor of inward radiance due to incident angle. Is the same as $\cos(\theta_i)$ where θ_i is the angle between ω_i and n

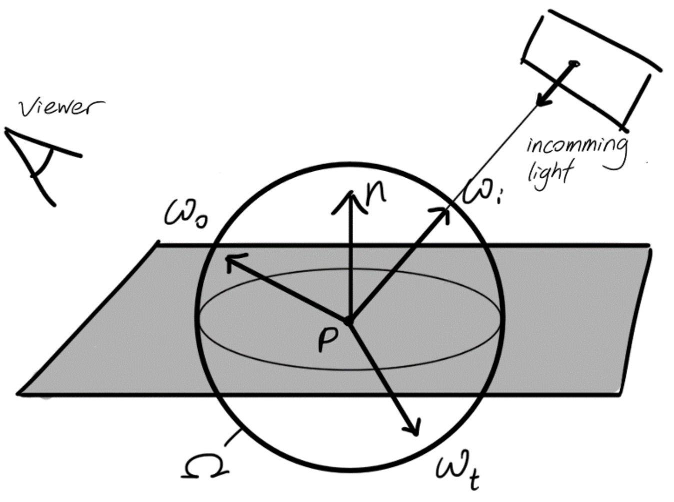

Figure 1: Terms used in the rendering equation.

The trick is to not be intimidated by it! There are two components to the sum: the emissive component $L_e(p, \omega_o)$, and the integral component. Let's take a deep look into who these two monsters are, I'm sure you'll find them to be quite tame.

1.1 The Emissive Component

Here's the smaller one of the two:

$$L_e(p, \omega_o)$$

This component is straight forward: It represents how much light is being emitted by the surface point p in the specified outgoing direction ω_o.

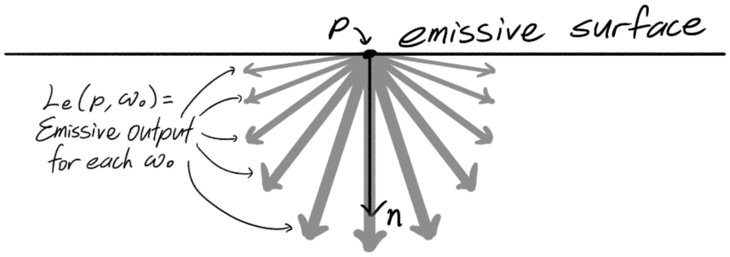

Figure 2: Representation of $L_e(p, \omega_o)$ for a hypothetical emissive surface.

Here's a question that one may ask: why can't L_e be a simple function of p instead of both p and ω_o? Does the amount of light emitted from a surface change for different outgoing directions? The answer is yes, and the reason why can be seen with some real world examples.

Chances are, we've all seen a laser pointer and a lightbulb sometime in our life. Both these things are emissive: they generate light when powered. Yet needless to say, the light generated from a laser is noticeably different from a lightbulb. The laser emits light in a straight, focused beam, while the lightbulb emits light in all directions. Now – for simplicity's sake – let's assume that the source of emission for both of these emissive objects comes from a surface that magically emits light. Given this, let's take surface points p_{laser} and $p_{lightbulb}$ from these two magical surfaces, and try to find their L_e component.

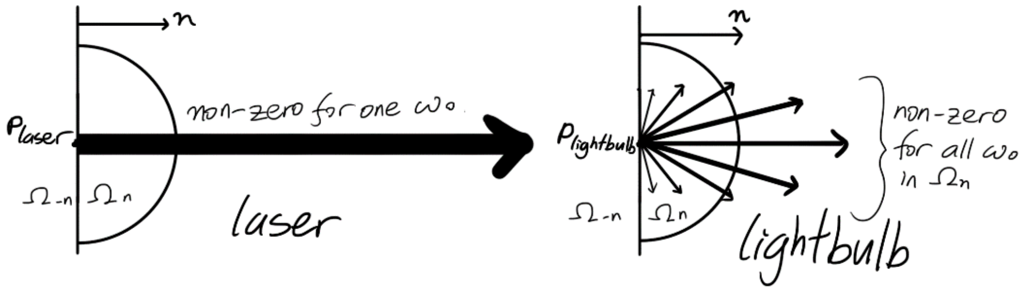

Figure 3: Emissive component for a laser vs. a lightbulb.

We see that $L_e(p_{laser}, \omega_o) \geq 0$ for a single direction ω_o only, and zero for all other ω_o's, while $L_e(p_{lightbulb}, \omega_o)$ will be non-zero for all directions ω_o in the half sphere on the side of the surface normal. Having L_e be a function of both p and ω_o not only enables us to represent both lasers and lightbulbs, but also anything in between. This is always a nice feature to have!

1.2 The Integral Component

A quick look at the real world will tell us that not all surfaces are emissive. We not only see objects that emit light, but also ones that reflect and transmit light. That is after all, how we see things: light from an emissive surface hits another surface, which reflects/transmits it to the direction of our eyes, which is then translated into the visual data we perceive.

The integral component represents the reflective and transmissive properties of surfaces, which when combined with the emissive property, gives us a full representation of a surface's visual properties.

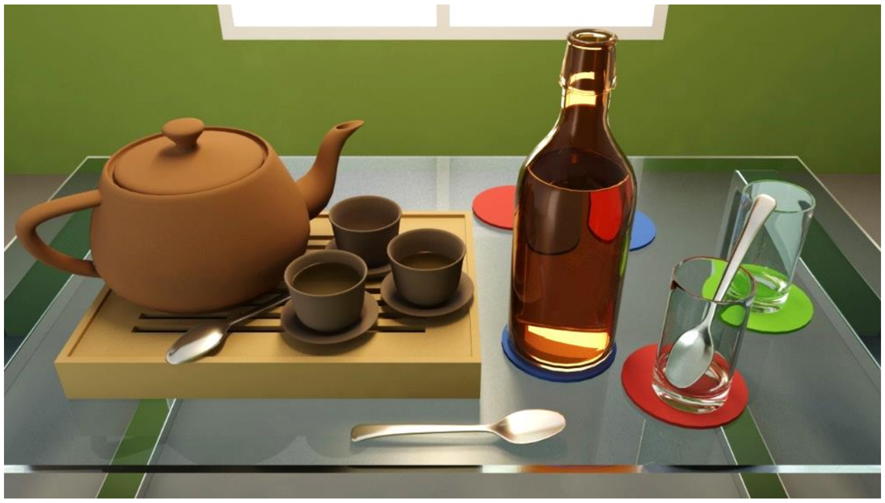

Figure 4: An image rendered using Mental Ray. Note that the window panes are modeled using an emissive surface. Other surfaces include clay (tea set), glass (table, bottles, and glass), and metal (spoons).

The integral component is deceivingly complex. What it states (or rather, asks) is simply:

> *"Given all possible incoming directions of light, what are their summed contribution to the specified outgoing direction?"*

To answer this question, let's simplify it to a single component of the sum:

$$f_s(p, \omega_i, \omega_o)\; L_i(p, \omega_i)\; (\omega_i \cdot n)$$

For an incoming direction ω_i, the amount of incoming light $L_i(p, \omega_i)$ needs to be found. After which, we multiply it by the weakening factor $(\omega_i \cdot n)$ (more on this later). Finally, we determine what proportion of $L_i(p, \omega_i)\ (\omega_i \cdot n)$ is *scattered* to the outgoing direction ω_o by multiplying it with the Bidirectional *Scattering* Distribution Function (BSDF) $f_s(p, \omega_i, \omega_o)$. The BSDF determines how the material will appear – whether it's matte, shiny, glossy, transparent, etc.

Scattering here refers to two components: reflection and transmission. For a material such as gold (or conductors in general), there is only reflection, since light cannot be passed (transmitted) through it. Yet for a material such as glass (or dielectrics in general), there exists both a reflective and transmissive component. These two components of the BSDF are called the BRDF (Bidirectional *Reflectance* Distribution Function) and the BTDF (Bidirectional *Transmittance* Distribution Function) respectively. Mathematically speaking:

$$f_s(p, \omega_i, \omega_o) = f_r(p, \omega_i, \omega_o) + f_t(p, \omega_i, \omega_o)$$

Where f_s, f_r, f_t represents the B**S**DF, B**R**DF, and B**T**DF respectively. Both $f_r(p, \omega_i, \omega_o)$ and $f_t(p, \omega_i, \omega_o)$ return a scaling factor. It tells us what proportion of light from ω_i should be reflected or transmitted into direction ω_o at the surface point p.

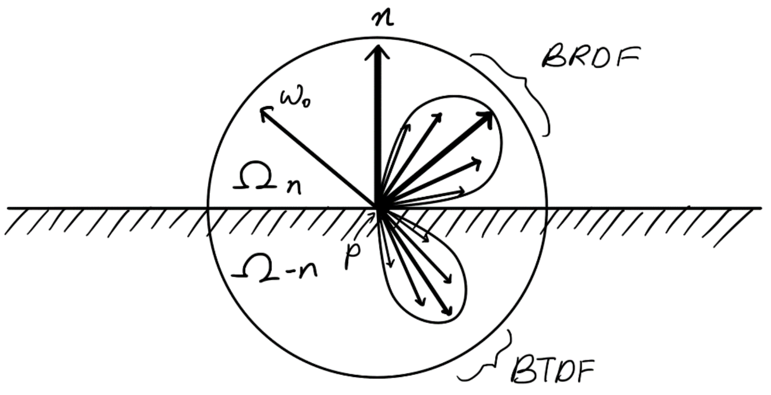

Figure 5: The BSDF (BRDF + BTDF) for a hypothetical dielectric surface. The BSDF here is represented as a set of ω_i directions that return a non-zero, positive value for fixed ω_o and p. The magnitude of each ω_i in this figure represents the magnitude of the returned value.

So how is $L_i(p, \omega_i)$ calculated? It isn't a function that can be solved directly, since incoming light can come from any point, on any surface within the scene. To determine what this surface point is, a recursive ray $r(p, \omega_i)$ must be sent into our scene with origin p and direction ω_i. The intersection point of this ray will be the surface point we're looking for.

Let the recursive ray's intersection point be $r_{ip}(p, \omega_i)$. Then we can define $L_i(p, \omega_i)$ as:

$$L_i(p, \omega_i) = L_o\big(r_{ip}(p, \omega_i), -\omega_i\big)$$

Where L_o is the rendering equation as defined above. Figure 6 explains pictorially why this is the case.

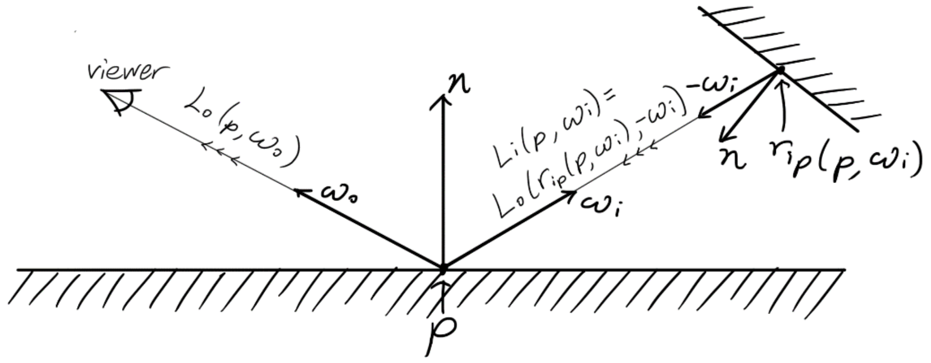

Figure 6: $L_o(p, \omega_o)$ represents the outgoing light from a surface point p in direction ω_o. Through some simple substitution, we can reformulate $L_i(p, \omega_i)$ as $L_o\big(r_{ip}(p, \omega_i), -\omega_i\big)$.

Why are we using L_o? Recall that $L_o(p, \omega_o)$ is "Radiance output at point p in direction ω_o." Therefore, $L_o\big(r_{ip}(p, \omega_i), -\omega_i\big)$ can be interpreted as "Radiance output at point $r_{ip}(p, \omega_i)$ in p's direction: $-\omega_i$", which is just another way of saying "Radiance input at p from direction ω_i." Substituting this into our single component of the sum, we have:

$$f_s(p, \omega_i, \omega_o)\ L_o\big(r_{ip}(p, \omega_i), -\omega_i\big)\ (\omega_i \cdot n)$$

Now let's take a look at what the weakening factor $(\omega_i \cdot n)$ aims to represent.

So far, we've considered "light" to be a unit-less line in 3D space consisting of a starting origin and a directional vector. In reality, light should be thought of as a bounded volume, with a unit of energy given for the light traversing through this bounded volume. This volume is created by forming a cone of a specified diameter, with the tip of the cone at the emission source, and the cap created by the intersection of this cone with an intersecting surface. If we hold the unit of energy constant, increasing the cap's diameter (and therefore surface area) will cause the energy contained to be spread over a larger area. Conversely, decreasing the cap's diameter will concentrate the energy contained over a smaller area. Here on earth, we see this effect taking place whenever we change seasons.

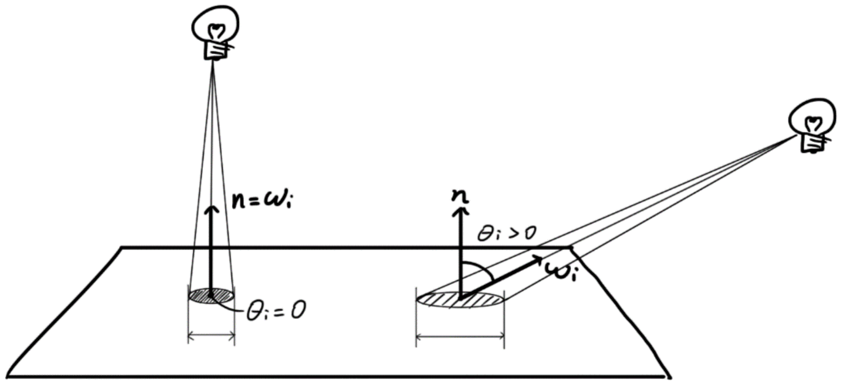

Figure 7: The cone from both light sources are identical in diameter, but since the light on the right is striking the surface on a glancing angle, its energy is distributed over a larger area. This reduces its intensity, which is noticeable on earth when the sun's rays hits our planet's surface at a glancing angle during the winter months.

This is where the weakening factor comes from. When this "light cone" hits a surface at a grazing angle, the cap of the cone is stretched out into an oval shape, increasing its surface area. Yet if the light cone hits a surface perpendicularly, the cap of the cone is at its smallest area possible. Since the energy within this cone is constant, an increase in surface area will spread out the contained energy, and therefore reduce its intensity. This effect is represented using $\cos(\theta_i) = (\omega_i \cdot n)$, which holds a value of 1 when the incoming direction ω_i is parallel to the surface normal n $(\cos(\theta_i = 0) = 1)$, and a value of 0 when ω_i is perpendicular to n $(\cos(\theta_i = \pi/2) = 0)$.

> ***Aside: A need for* quantities*.***
>
> *You may now realize that there's a need for units to describe the quantities of light energy. So far, this has been completely glossed over to keep things simple. If having an explanation with detailed descriptions of quantities is preferred, consider taking a look at the units used in radiometry[ii], and Kajiya's paper[i].*

Extending the single component of the sum to the integral is simple. The integral considers all possible incoming directions ω_i, summing the single components for each ω_i together to get the total contribution of all incoming light. The final recursive rendering equation we aim to compute is then:

$$L_o(p, \omega_o) = L_e(p, \omega_o) + \int_\Omega f_s(p, \omega_i, \omega_o)\, L_o\big(r_{ip}(p, \omega_i), -\omega_i\big)\, (\omega_i \cdot n)\, d\omega_i$$

Equation 1: The recursive rendering equation.

2 Pathtracing – Computing the Rendering Equation

Although the concept of the rendering equation is simple, computing its integral component is no easy task. We have to account for an infinite number of possible ω_i directions from within the hemisphere. The bad news that an exact solution is unfeasible to find. The good news is we can get very close to it!

Pathtracing is one of the rendering techniques used to numerically solve the rendering equation. Most commonly, it is used in combination with Monte Carlo integration, which is a numerical integration technique used to compute the definite integral of a function using random sampling[iii]. For the purposes of our pathtracer, we can use it to approximate the value of the recursive integral component within the rendering equation.

Aside: Some Definitions

Before going into details, let's define some of the terms we'll be using. When we are talking about a ***ray path****, it is the path that a single ray takes from its creation, to its termination. It is composed of* ***ray segments*** *that span between two consecutive* ***surface points*** *within the ray path.*

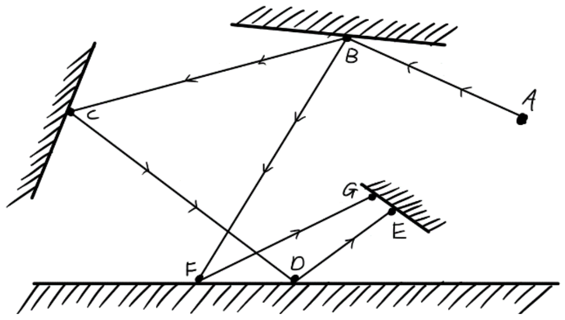

Figure 8: There are 2 ***ray paths*** *in this figure: ABCDE, and ABFG. There are 6* ***ray segments*** *in this figure: AB, BC, CD, DE, BF, and FG. There are 7* ***surface points*** *in this figure: A, B, C, D, E, F, and G.*

What makes pathtracing special when compared to other ray tracing methods is how it handles the recursive part of the integral. For example, in distributed ray tracing, the recursive integral is fully computed up to a certain accuracy before returning its results up one recursive level. In pathtracing, only a single component of the sum is calculated before returning. At any given time, pathtracing only considers one ray path, versus a distributed ray tracer that considers a "fan" of all possible ray paths.

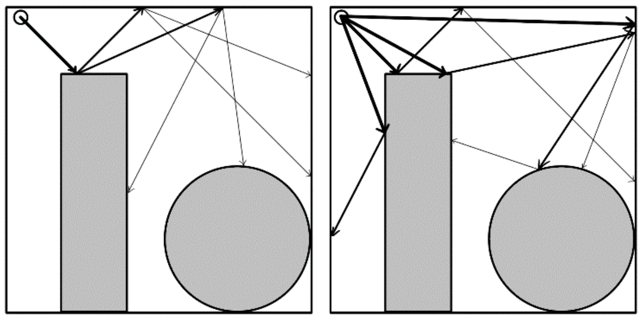

Figure 9: The tracing approach of a distributed ray tracer (left), vs. a pathtracer (right) using a max depth of 2. Both are tracing 4 ray paths. Initial rays are in bold, and rays of increasing depth get progressively thinner. A distributed ray tracer traces a "fan" of paths for each initial ray, while the pathtracer traces a single path per initial ray.

Mathematically speaking, any ray tracer approximating Equation 1 using Monte Carlo integration will do it like so (discussed in detail in 2.1) :

$$L_o(p,\omega_o) \approx L_e(p,\omega_o) + E\left[\frac{f_s(p,\omega_i,\omega_o)L_o\big(r_{ip}(p,\omega_i),-\omega_i\big)(\omega_i \cdot n)}{pdf(\omega_i)}\right]$$

Where $E[X]$ gives the expected value of a random variable X – a fancy way of saying the long-run average. In the case of a distributed ray tracer, we can take as many samples as we like to calculate this expected value. But for a path tracer, we're only allowed to take one sample. The long-run average of a single sample is – you guessed it – just the value of the sample itself:

$$L_o(p,\omega_o) \approx L_e(p,\omega_o) + \frac{f_s(p,\omega_i,\omega_o)L_o\big(r_{ip}(p,\omega_i),-\omega_i\big)(\omega_i \cdot n)}{pdf(\omega_i)}$$

Equation 2: The rendering equation for pathtracing using Monte Carlo integration with importance sampling.

The reasoning behind tracing only one path has to do with a ratio: The number of ray paths traced, versus the magnitude of their contributions to the rendered image. Ideally, we wish to keep this ratio as low as possible; to have each ray path give a maximum amount of contribution to the final rendered image. Computation time is a resource, and by rationing it to the rays that does the most work, we can make our ray tracer more efficient. A pathtracer aims to do this by reserving computation time for ray segments of low recursive depth, as they contribute the most to the final rendered image. Figure 10 provides empirical evidence towards this fact. Note that this does not mean that a pathtracer ignores higher depth ray segments, but rather, it traces more ray segments of a lower depth when compared to other ray tracing techniques.

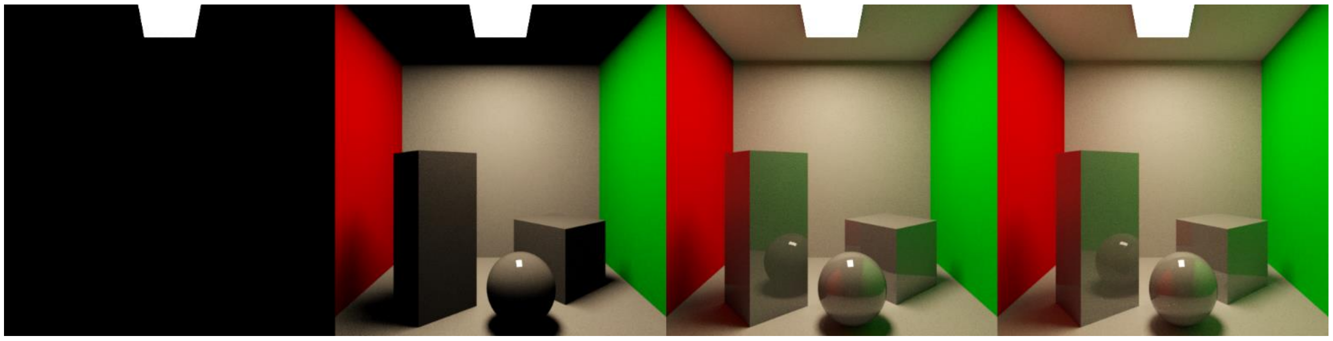

Figure 10: Results of limiting the maximum ray depth. The max depths are 0, 1, 2, and 3 going from left to right respectively.

So why would tracing a single ray path be more efficient than tracing a fan of paths? In Figure 9, we see a difference in the amount of low-depth segments (bolder lines) despite tracing the same number of paths. More specifically, the pathtracer considers more low-depth segments than the distributed ray tracer. Tying this in to what we see in Figure 10, we see that the pathtracer is more efficient, as it allocates more resources to initial segments that contribute the most to the final image. Also, note that a pathtracer can choose to terminate the ray path early, while it is still below the maximum depth. This is demonstrated on the left-most ray for the path tracer in Figure 9. This feature, named "Russian roulette" (2.2), allows pathtracers to dedicate less resources to higher depth segments, and improve efficiency even further.

2.1 Monte Carlo Integration

Monte Carlo integration is a technique used in a variety of fields. For the purposes of our pathtracer, it is used to approximate the entire integral component of the rendering equation with just one sample of the function being integrated.

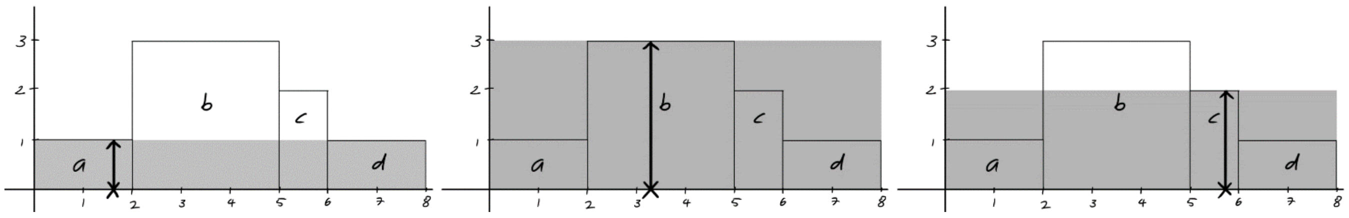

Figure 11: Three separate attempts to sample a piecewise function in order to compute its integral.

Consider the simple piecewise function $f(x)$ above. Its domain is in the range of $[0,8)$, and its integral across its domain is $1 * 2 + 3 * 3 + 2 * 1 + 1 * 2 = 15$. We would never actually use Monte Carlo integration on such a simple function, but for the same of demonstration, we will see how it can be done.

Let's take a look at the three sample attempts in Figure 11. The three samples are generated in a uniformly random fashion within the bounds of the domain. The first sample landed somewhere between 1 to 2, and the function evaluates to 1 at that point. With this single sample, and no knowledge of how the rest of the function behaves, the best we can do is to scale the sample up by multiplying it with a *weight*. In this case, the weight is equal to the size of the domain (which is 8), giving an approximate result of 8 for this sample (more on *weights* later). For the second and third sample, we take the same approach. The function evaluated at the samples taken are 3 and 2 respectively, and multiplying them by the weight, we get 24 and 16.

All three results from the samples are within a reasonable range of 15, but we can get a better result if we average them: $(1 * 8 + 3 * 8 + 2 * 8) / 3 = 16$, which is not too far off from the expected result of 15. Pretty good for only taking 3 samples!

Since we're talking about picking samples, let's make this more robust by adding some probability into the mix. Let X be a continuous random variable from which the samples are picked from. Probability $P\big(X \in f's\ domain\ [0,8)\big) = 1$, and anything outside of f's domain has probability 0. Because X is uniformly distributed, the probability density function of X is:

$$pdf_X(x) = \begin{cases} 1/8\ if\ x \in [0,8) \\ 0\ otherwise \end{cases}$$

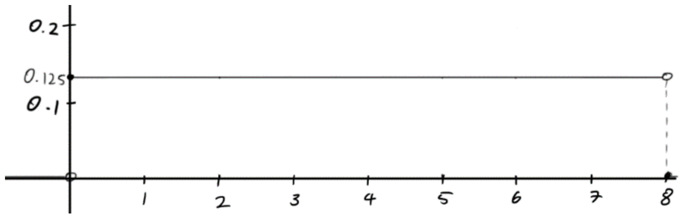

Figure 12: A pdf that gives the probability of each sample being taken within a given range. Note that the pdf must integrate to 1.

Using our sampling technique as above, we can derive the following formula to approximate the integral:

$$\int f(x)\,dx \equiv E[f(y) * weight]$$

Where y is a sample drawn from X, and $E[Z]$ is the expected value of random variable Z.

As an example, we can apply this to the three samples taken. The expected value here is identical to the average of the three weighted samples. Let's say we sampled 1.6, 3.1, and 5.8 respectively, as depicted in Figure 11:

$$\begin{aligned}\int f(x)\,dx &\approx (f(1.6) * weight + f(3.1) * weight + f(5.8) * weight)\frac{1}{3}\\ &\approx (1 * 8 + 3 * 8 + 2 * 8)\frac{1}{3}\\ &\approx 16\end{aligned}$$

So what exactly is this weight? Intuitively, it tells us how much a specific sample should contribute to the expected value. It isn't some magical number, but rather a very specific one. The weights must be normalized in such a way, where if the expected value is calculated using an indefinite number of samples, then it will give the exact result of the integral being estimated. Without diving into proofs, this weight happens to be the inverse of the pdf for the random variable from which the samples are drawn. In the case of uniformly random X, it is pdf_X^{-1} as seen in Figure 12, or simply 8 for x within domain [0, 8).

Knowing this, we can rewrite our estimator as

$$\begin{aligned}\int f(x)\,dx &\equiv E\left[\frac{f(y)}{pdf_X(y)}\right]\\ &\equiv \frac{1}{n}\sum_{i=1}^{n}\frac{f(y_i)}{pdf_X(y_i)}\end{aligned}$$

Equation 3: The Monte Carlo estimator for n samples. pdf$_X$ *is the probability density function of random variable* X *from which our samples are picked.*

This estimator is in fact, the Monte Carlo estimator. To understand this further, let's construct a different random variable X' to use for sampling f and see what differences it will cause:

$$pdf_{X'}(x) = \begin{cases} 0.175 \; if \; x \in [2, 5) \\ 0.095 \; if \; x \in [0, 2) or \; x \in [5, 8) \\ 0 \; otherwise \end{cases}$$

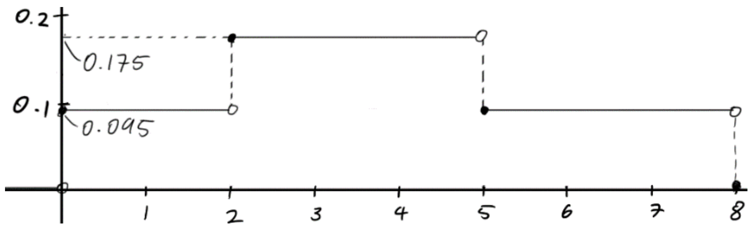

Figure 13: The new pdf, giving a higher probability to samples within the range [2, 5).

By sampling X' instead, the distribution of our samples is now specified by $pdf_{X'}$. More specifically, the location of our samples will now favor the range [2, 5) over others due to a higher probability being assigned by $pdf_{X'}$. As a concrete example, our three samples from Figure 11 may not be 1.6, 3.1, and 5.8

anymore, but rather 1.6, 3.1, and 4.8. This also changes the weight of each sample. Using our set of new samples picked from X', the approximation is now:

$$\begin{aligned}\int f(x)\,dx &\approx \left(\frac{f(1.6)}{pdf_{X'}(1.6)}+\frac{f(3.1)}{pdf_{X'}(3.1)}+\frac{f(4.8)}{pdf_{X'}(4.8)}\right)\frac{1}{3}\\ &\approx \left(\frac{1}{0.095}+\frac{3}{0.175}+\frac{3}{0.175}\right)\frac{1}{3}\\ &\approx 14.94\end{aligned}$$

Which is surprisingly close to the expected result of 15.

You may have now realized that we are free to pick the pdf to define the random variable in which we are sampling from. In the case of $pdf_{X'}$, we've decided to give the range [2, 5) almost twice the probability of being sampled in comparison with the rest. As a result, in order to ensure that each sample has a fair contribution to the expected value, we have to weight samples within [2, 5) almost half as less as other samples. This is already being done through the use of $pdf_{X'}(y)^{-1}$ as the weight.

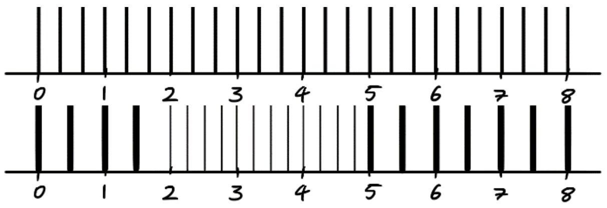

Figure 14: On top, samples are picked using uniform X. On the bottom, samples are picked using our modified X'. Sample density changes once we sample from X'. Therefore, in order to give each sample a fair contribution, low-probability samples are given more weight within the expected value (bolder), and higher-probability samples are given less (lighter).

2.1.1 Importance sampling

You'll also have realized that our choice of pdf has given us a better approximation of the integral. The technique of picking a pdf that will give us a better estimation is called *Importance sampling*. In fact, the Monte Carlo estimator given in Equation 3 is not the general form, but one with importance sampling included. The use of a probability density function to serve the dual purpose of determining the density of samples, and their weights within the expected value, is what adds this feature.

$$E\left[\frac{f_s(p,\omega_i,\omega_o)L_o\big(r_{ip}(p,\omega_i),-\omega_i\big)(\omega_i\cdot n)}{pdf(\omega_i)}\right]$$

Equation 4: The Monte Carlo estimator with importance sampling used to approximate the rendering equation's integral.

The questions is then "how should we pick this pdf?" Intuitively, we wish to use a pdf that picks more ω_i samples within the hemisphere that will give the current ray path a higher contribution. However, the behavior of the term $L_o\big(r_{ip}(p,\omega_i),-\omega_i\big)$ within the estimator is a wild card, as it will be different for each surface point. This makes coming up with an ideal pdf difficult. The solution to this is to stick our heads into the sand, and ignore the problem.

Because $L_o\big(r_{ip}(p,\omega_i),-\omega_i\big)$ is being multiplied by $f_s(p,\omega_i,\omega_o)(\omega_i\cdot n)$, which are functions that we know the behavior of, we can base our pdf upon them instead. How much $L_o\big(r_{ip}(p,\omega_i),-\omega_i\big)$ will contribute is directly affected by the value of $f_s(p,\omega_i,\omega_o)(\omega_i\cdot n)$, making our lazy problem solving approach a good idea after all. Still, finding the right pdf using $f_s(p, \omega_i, \omega_o)(\omega_i \cdot n)$ is fairly involved. Walter & Torrance's paper on microfacet BRDF models[iv] presents a method to do so.

2.2 Russian roulette

Pathtracing, despite having a maximum recursion depth for ray paths, implements a heuristic called "Russian roulette" to terminate ray paths early. It improves efficiency by preventing the pathtracer from wasting computational resources on ray segments that have minimal contribution. It is executed at every recursive call, and it's returned result states whether the current ray path should continue on its path, or terminate immediately. How this decision is made is up to the programmer. Most frequently, a termination probability is calculated based upon the current ray segment's estimated intensity. An inverse relationship is made, where a higher estimated intensity will result in a smaller termination probability, and a lower estimated intensity will result in a higher probability.

2.3 Explicit sampling

Explicit sampling is another optimization technique used to increase the magnitude of contribution for each ray path. It involves explicitly sampling an extra ray segment at each surface point that is pointing towards an emissive surface. This ensure the capture of direct illumination, and increases contribution.

The reasoning behind this is simple: Emissive surfaces are what provides illumination for all the surfaces within the scene. If we, through the random sampling of ω_i, never hit an emissive surface along a ray's path, then by definition, the rendering equation for this ray path returns 0. This would result in this path providing zero contribution to the final image.

Explicit sampling solves this issue by splitting the main ray segment into two. One goes on to complete the main ray path, while the other goes in the direction of a chosen emissive surface. The latter is the explicit sample, and it ensures that all ray paths will consider emissive surfaces. This minimizes the number of ray paths that return 0.

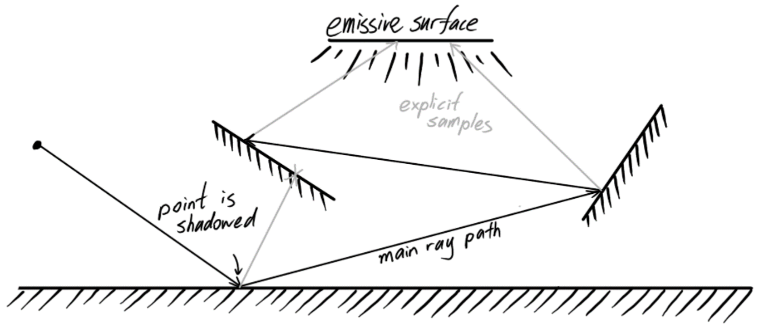

Figure 15: Grey ray segments are the explicit samples. They fork out of each surface point within the main ray path, in the direction of an emissive source.

Note that doing this does not technically break the law of pathtracing. The ray segment generated for explicit sampling is not recursive, and is only used to determine whether the surface point is being shadowed by another surface. The point is shadowed if there is an interfering surface between the point and the emissive surface, resulting in an emissive contribution of 0. Otherwise, the emissivity of the emissive surface hit is noted. In both cases, the resulting contribution is considered as a second sample within the expected value of the Monte Carlo estimator.

However, there are a few technicalities to resolve. One has to do with the wrongful inclusion of extra emissive terms, and the other with importance sampling.

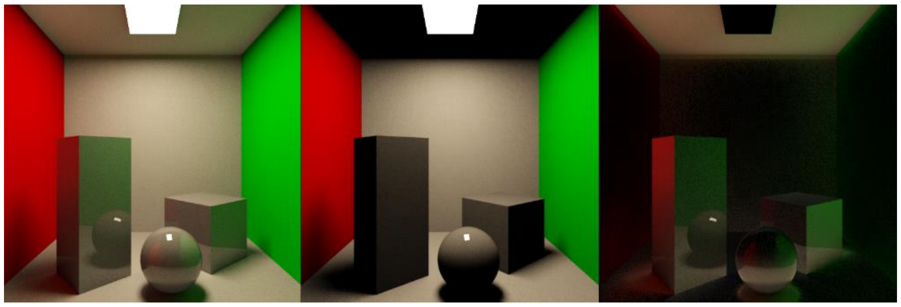

Figure 16: For easy reference, we'll name these pictures A, B, C going from left to right. A and B are taken from Figure 10, and are the results of setting the maximum recursion depth to 2 and 1 respectively. 3 is the result of subtracting B from A. It is the indirect illumination component by itself.

In reality, what explicit sampling is doing is *splitting* the recursive integral component into two parts: a direct illumination component, and an indirect illumination component. The effects of these two components can be seen in Figure 10, where a max depth of 1 returns the direct illumination only, and any depths beyond that returns both the direct and indirect illumination. We can see the indirect component by itself when we subtract the direct component from the combined output, as shown in Figure 16. Intuitively, if we were to consider direct illumination separately with an explicit sample, then we have to ensure that the recursive call on the main path doesn't consider it a second time. This is done by removing the emissive component L_e from the recursive call on the main path, and instead consider it using the explicit sample at the current recursive level.

Lastly, if importance sampling is used, then each ray segment must be sampled according to the chosen pdf. Yet the explicitly sampled segment isn't sampled according to the pdf, but is sampled by picking a random emissive surface from within the scene. Therefore, the weight to use here isn't the inverse of the pdf, but a constant term in relation with the number of emissive surfaces within the scene. There are a few possible weights to use, but for the D95 pathtracer, the weight is simply the number of emissive surfaces within the scene.

3 The Pathtracing Algorithm

Despite having a relatively simple algorithm, a pathtracer is very powerful. It can replicate a wide spectrum of lighting effects, with highly scalable quality settings to create anything from quick test renders, to production quality renders. The general pathtracing algorithm is widely agreed upon, but for this section, we'll be specifically going through the algorithm used in the D95 pathtracer.

A pathtracer requires a few things to render a scene:

1. Camera location and settings
2. Scene containing geometry to render
3. At least one emissive surface to provide illumination for the scene
4. Number of samples per pixel
5. Maximum recursion depth.

Once these things are in place, we are ready to generate ray paths. We will first start with a very high-level overview, then gradually dig deeper on each of the bolded points.

```
Fetch arguments for samples per pixel, and max recursion depth;
Initialize camera;
Initialize scene;
Initialize output image array;
Render image;
Save to output image;
Done;
```

This is the outline of any raytracing algorithm. How the camera and output images are initialized are up for the programmer to decide. We'll dig slightly into how the scene is initialized, and then finally dig deep into how the image is rendered.

The scene's geometry within the D95 pathtracer are represented using triangle meshes. This data is stored in the Wavefront OBJ format[v].

```
INITIALIZE SCENE

Load triangles into array:
    Load verticies;
    Load texture coordinates;
    Load normals;
    Load material properties reference;
Create array referencing emissive triangles;
Load material properties into array:
    Load texture maps, if referenced;
Construct octree spatial subdivision structure;
```

After the scene and camera have been initialized, we are ready to start tracing rays.

```
RENDER IMAGE

For each pixel in image:
    For each sample out of specified # of samples numSamples:
        r = Initialize a ray going through a point within the current pixel;
        resultColour += Compute the rendering equation using (r);
    Average the result = resultColour /numSamples;
    Save the final value of pixel to image, capping within range 0-255 inclusive;
```

Here, we see how the quality of the rendered image is defined. The number of samples – as specified by the user – determines how many ray paths will be traced per pixel, and directly effects the output quality. In addition, different pathtracers may decide differently on how to distribute the samples across the pixel. Some may distribute it evenly, while others may distribute it in a uniformly random fashion.

```
COMPUTE THE RENDERING EQUATION using (r)

Send ray r into scene, and identify intersection point;
iData = Collect intersection data;
Initialize return colour L_o = L_e = Emissive colour @ p;
If max recursion depth reached, or russian roulette decides to terminate:
    Return L_o;
isSameHemisphere = Determine if intersection occurred on same side of surface normal;
r1, r2 = generated random doubles;

Pick w_i and calculate BSDF using (r1, r2, p, w_o = -direction of ray r, iData):
    isTransmitted = whether BSDF chose to reflect or transmit the ray;
    w_i = the direction picked;
    bsdfVal = value returned by BSDF;

integralComponent = Initialize to 0;
If bsdfVal > 0: //implies non-zero integral component
    Initialize a ray r' = r with origin = p and direction = w_i;
    Modify Index of Refraction stack of r' using (isTransmitted, isSameHemispehre);
    L_in = Compute the rendering equation (RECURSION) using r';
    integralComponent = bsdfVal * L_in * (1/cos(theta_i));
Return L_o = L_o (initialized to L_e) + integralComponent;
```

To help connect with the rendering equation, the related terms have been highlighted. Note that a few specific performance enhancing features have been left out from this outline, namely explicit light sampling (2.3), and importance sampling (2.1.1). The pathtracer still works perfectly fine, but will generate noisier results in comparison.

This outline marks the end of the high-level description of a path tracer. The contents below will expand further on what is being done at each point.

When an intersection point is found, we need to collect all the information necessary to compute the rendering equation at that point:

```
COLLECT INTERSECTION DATA

Collect the following:
    Intersection point coordinate p;
    Normal of surface @ p;
    Texture coordinate of surface @ p;
    BSDF & material colours of surface @ p;
Return collected data;
```

Before we jump into picking the ω_i direction and computing the BSDF, we have to check whether the current ray path should terminate. This condition is determined by the specified maximum recursion depth, and the Russian roulette algorithm. If either or decides to terminate the path, we return the emissive color of the surface at p, forming the base case of the recursion.

If we decide to continue down this ray path, a ω_i direction for the recursive ray is sampled, and the value of the BSDF is calculated before sending the recursive ray:

```
PICK w_i AND CALCULATE BSDF using (r1, r2, p, w_o, iData)

w_i = Using r1, r2, sample w_i according to sampling technique;
bsdfVal = Result of computing BSDF with p, w_i, and w_o:
    Using the collected iData, compute the BSDF of p;
isTransmitted = whether ray was reflected or transmitted;
Return isTransmitted, w_i, bsdfVal;
```

Aside: "Where am I?" asks the ray.

There is a small technicality to consider before sending the recursive ray. To property calculate transmission, we need to know the IOR (index of refraction) of the bounded volumes on both sides of the surface. Since our ray path can traverse through many transmissive surfaces that – although physically impossible – are intersecting each other, we have to keep track of the IOR as the path progresses.

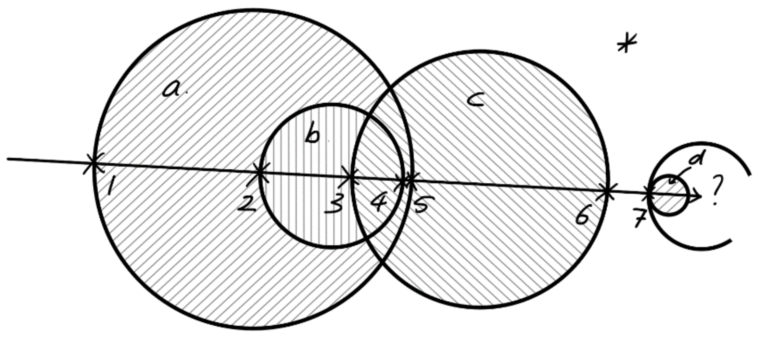

*Figure 17: A ray going through five transmissive objects. The shaded areas are bounded volumes created by the geometry, and are labeled with a letter. The space containing all the geometry is labeled as *. The intersection points of the ray with the geometry are marked with x's, and are numbered.*

Of course, this wouldn't be needed if we simply restrict our scene's geometry to never intersect each other. You could do that, but then things wouldn't be crunchy now, would they?

To make matters worse, the two objects that are at intersection point 7 can exist within our scenes. Object '?' isn't closed, so it doesn't bind any space, and object 'd' merges with '?' at point 7. The question we need to answer here is "What is the IOR of the object at 7, and will there be a guaranteed exit point?" The answer is an unsatisfying "I don't know", or more truthfully, "I can't be bothered to find a solution for this." The good news is that many commercial ray tracers can't be bothered either.

To keep track of the IOR's of volumes a, b, and c, we implement a stack data structure within the ray. Each time the ray enters or exits a surface, we modify the stack to reflect the ray's current IOR state:

```
MODIFY INDEX OF REFRACTION STACK OF r' using (isTransmitted, isSameHemispehre)

// See Figure 18 for each case.
Switch (isTransmitted, isSameHemisphere):
    Case (false, false): Total internal reflection. No need to modify stack. (a);
    Case (false, true): Reflection. No need to modify stack. (b);
    Case (true, false): Transmission out of object. Remove from stack. (c);
    Case (true, true): Transmission into object. Add to stack. (d);
```

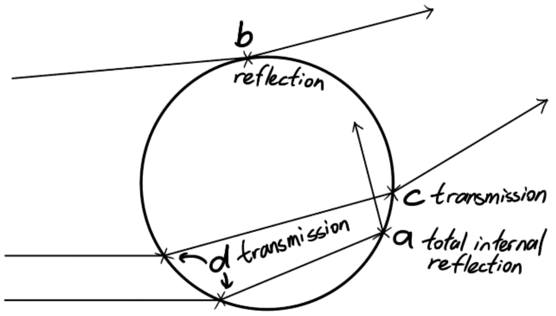

Figure 18: Different ways a ray can behave when interacting with a transmissive object.

After modifying the IOR stack, we are ready to send the recursive ray. The rendering equation is recursively calculated, fetching the value of L_{in}. As defined in the rendering equation, we multiply L_{in} with the BSDF value and weakening factor to get the estimate of the integral component. The emissive component L_e is added, and the result is returned.

This marks the end of tracing one single ray path. The process is repeated for each sample within each pixel. The final image is then saved in a location as specified by the user.

4 REFERENCES

[i] Kajiya, J. T. (1986). The Rendering Equation. In Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques (pp. 143–150). New York, NY, USA: ACM. http://www.cse.chalmers.se/edu/year/2011/course/TDA361/2007/rend_eq.pdf

[ii] Mallet, I. (2016). The Idiots' Guide to Radiometry and Colorimetry Page. Retrieved April 25, 2016, from http://geometrian.com/programming/tutorials/radcol/index.php

[iii] Monte Carlo integration. (2016, February 8). In Wikipedia, the free encyclopedia. Retrieved from https://en.wikipedia.org/w/index.php?title=Monte_Carlo_integration&oldid=703959663

[iv] Walter, B., Marschner, S. R., Li, H., & Torrance, K. E. (2007). Microfacet Models for Refraction Through Rough Surfaces. In Proceedings of the 18th Eurographics Conference on Rendering Techniques (pp. 195–206). Aire-la-Ville, Switzerland, Switzerland: Eurographics Association. http://www.cs.cornell.edu/~srm/publications/EGSR07-btdf.pdf

[v] Wavefront .obj file. (2016, April 25). In Wikipedia, the free encyclopedia. Retrieved from https://en.wikipedia.org/w/index.php?title=Wavefront_.obj_file&oldid=717037921