Global Illumination and Path Tracing

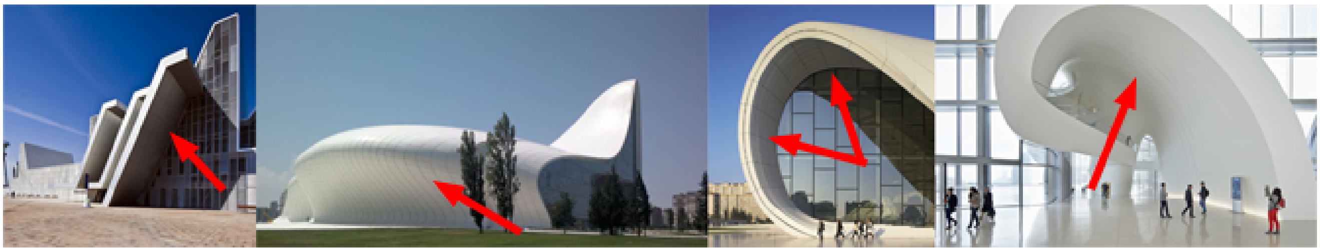
Contents

Keywords: global illumination, light transport algorithms, indirect diffuse, indirect specular, caustics, Monte Carlo integration, path tracing, Monte Carlo path tracing, backward tracing, forward tracing, bidirectional path tracing, photon mapping, sampling, radiance, noise, distant environment light, image based lighting.

In this lesson we will use Monte Carlo methods to simulate global illumination. It will be hard if not impossible to understand this lesson fully if you are not familiar with Monte Carlo methods. Hopefully for you, we have covered this topic extensively in the section The Mathematics of Computer Graphics. Read these lessons first before proceeding.

What is Global Illumination Anyway?

Global illumination is actually pretty simple to simulate. Though generally, the main or principal issue with global illumination is that no matter what you do, it is generally very expensive to simulate and difficult to simulate efficiently.

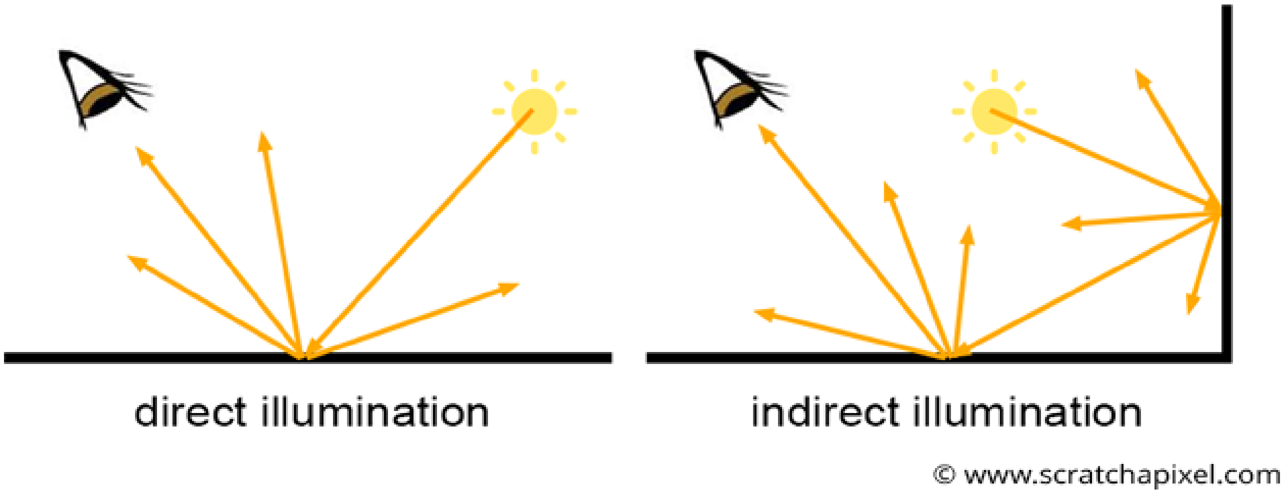


But what is global illumination in the first place anyway? As you know, we see things because light emitted by light sources such as the sun bounces off of the surface of objects. When light rays bounce only once from the surface of an object to reach the eye, we speak of **direct illumination**. But when light rays are emitted by a light source, they can bounce off of the surface of objects multiple times before reaching the eye. This is what we call **indirect illumination** because light rays follow complex paths before entering the eye. This effect is clearly visible in the images above. Some surfaces are not exposed directly to any light sources (often the sun), and yet they are not completely black. This is because they still receive some light as an effect of light bouncing around from surface to surface.

So far, we learned about simulating direct illumination. Global illumination involves to simulate both effects: direct plus indirect illumination. Simulating both effects is important to produce realistic images.

Being able to simulate global illumination seems like a feature that every rendering system should have so why making a big deal out of it? Because while simple in some cases, it is complex to solve in a generic manner and also generally expensive. This explains why for example most real-time rendering systems don't offer to simulate (at least accurately) global illumination effects.

- It is hard to come with a generic solution that solves all light paths: light rays can interact with many different kind of materials before entering the eye. For example a light ray freshly emitted from a light source may be reflected by a diffuse surface first, then travel though a glass of water (it has been refracted then which means a



slight shift or change of direction), then hit a metal surface, then another diffuse surface and then finally hit the eye. There is an infinity of possible combinations. The path that a light ray follows from the light source to the eye is simply called a **light path**. As you know, in the real world, light travels from light sources to the eye. From a mathematical point of view, we know the rules that define the way light rays are reflected off from the surface of various materials. For example we know about the law of reflection, we know about refraction, etc. Simulating the path of a light ray as it interacts with various materials is thus something that we can easily do with a computer. Though the problem as you know, is that **forward tracing** in computer graphics is not an efficient way of making up an image. Simulating the path of light rays from the light source they are emitted from to the eye is inefficient (see the lesson Introduction to Ray Tracing: a Simple Method for Creating 3D Images). We prefer **backward tracing** which consists of tracing the path of light rays from the eye, back to the light source where they originated from. The problem is that backward tracing is an efficient way of computing direct illumination indeed but not always an efficient way of simulating indirect lighting. We will show why in this lesson.

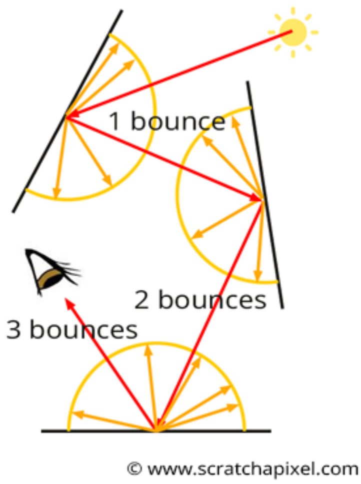


Figure 1: an example of forward tracing. A light ray may bounce many times before entering the eye.

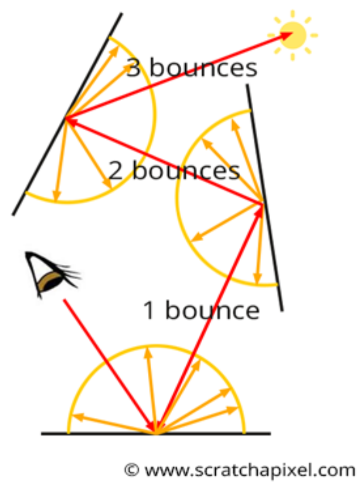


Figure 2: simulating indirect diffuse using backward ray-tracing.

- Direct lighting mostly involves to cast shadow rays from the shaded point to the various light sources contained in the scene. How fast computing direct lighting is depends directly on the number of light sources contained in the scene. Though computing indirect lighting is generally many times slower than computing direct lighting because it requires to simulate the path of a great number of additional light rays bouncing around the scene. Generally, simulating these light paths can be more easily done with rays and ray-tracing though ray-tracing as we also know is quite expensive. As we progress with the lesson, understanding why simulating global illumination is an expensive process will become clearer.

What you need to remember from this part of the lesson, is essentially that there is two ways for an object to receive light: it can receive light directly from a light source or indirectly from other objects. Objects in a way can also been seen as light sources. Not because they emit light though but because they reflect light. Thus we need to take them into consideration when we compute the amount of light a shaded point receives (of course, to reflect light these objects need to be illuminated themselves somehow either directly or indirectly). You may guess that this is a complex problem on its own because light reflected by an object A towards an object B might be reflected by B back to A, which in turns will reflect some of that light again back to B, etc. The lighting of B depends on A, but the lighting of A also depends on B (figure 3). The process implies an **exchange of energy between surfaces** which can go for ever. Putting this in mathematical form is possible, but solving the resulting equations is much harder (for a more formal introduction to this topic check the lesson devoted to the **Rendering Equation** in the next sections).

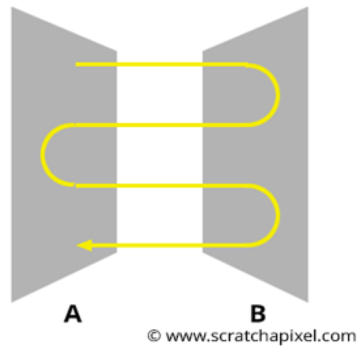


Figure 3: exchange of light energy between two unoccluded patches. Illumination of B depends on the light reflected or emitted by A in the direction of B, but some of that light is reflected back to B which reflects it again to A, etc.

Simulating indirect illumination is about simulating the paths that light rays take from the moment they are emitted by a light source to the moment they enter the eyes. For this particular reasons we generally speak of **light transport**, the transport of light or its propagation in space as it bounces from surface to surface. A **light transport algorithm** (backward tracing which we will talk about again this chapter is an example of such algorithm) is an algorithm that describes how this light transport problem can be solved. As you we will explain in this lesson, simulating light transport is a complex problem, and it is hard to come with one algorithm that makes it possible to describe all sort of light paths that we find in the real world.

Please refer to the chapter Light Transport from the lesson Rendering an Image of a 3D Scene: an Overview. It contains a more complex introduction to the concept of light transport and it presents a method proposed by Paul Heckbert to describe light paths using regular expressions.

How Do We Simulate Indirect Lighting with Backward Tracing

To understand the problem of simulating global illumination with backward tracing, you first need to understand how we can simulate indirect lighting. As suggested, many attempts have been made to formalize this process i.e. putting the process into a mathematical form (interested and mathematically savvy readers can fine some information in the reference section at the end of this lesson). Though, we will first explain the process in simple terms. If you are interested in the maths, please check the advanced lessons on rendering in the next sections.

We know from the previous lesson that light illuminating a given point P on the surface of an object, can come from all possible directions contained within a hemisphere oriented about the surface normal N at P. When you deal with direct light sources, finding where light comes from is simple. We just need to loop over all the light sources contained in the scene and either consider their direction if it is a directional light or trace a line from the light source to P if the light is a spherical or point light source. Though when we want to account for indirect lighting, every surface above P may redirect light towards P. Though, as we just mentioned, finding the direction of light rays when you deal with delta lights is simple but how do we do that when light is emitted by a surface? There is no unique point light position from which the light direction can be computed. In fact, since we deal with a surface, there is an infinity of points we could chose on the surface of that object to compute the light direction. And why should we choose one direction rather than another considering that every point on the surface of that object is likely to reflect light towards P?

The answer to this question is both simple and difficult. In some very specific situations, the problem can be solved analytically, there is a closed-form solution to this problem. In other words, it can be solved with an equation. This happens when the shapes of the objects making up the scene are simple (sphere, quads, rectangles), when the objects are not shadowing each other and when they have constant illumination across their surface. As you can see this makes a lot of "if". In other words, it is for example quite simple to compute the exchange of light energy between two rectangular patches or between a rectangle and a sphere as long as they are not **occluded** by some other objects and as long as they have a constant color. But as soon as we place a third object between the patch and the sphere or as soon as we shade these objects, then these simple analytical closed-form solutions don't work anymore. To find more information on this topic please check the lessons on rendering in the next sections (and especially the lesson on radiosity).

We thus need to find a more robust solution to this problem in which the solution works regardless of the objects shapes and doesn't depends on objects visibility. It turns out that solving this problem with these constraints is pretty hard. Hopefully we can use a simple method called Monte Carlo integration. The theory behind Monte Carlo integration is explained in great detail in two lessons you can find in the section The Mathematics of Computer Graphics. But why **integration**? Because in essence what we are trying to do is "gather" all light coming from all possible directions above the hemisphere oriented about the normal at P and this in mathematics, can be done using what we call an "integral" operator:

$$\text{gather light} = \int_\Omega L_i.$$

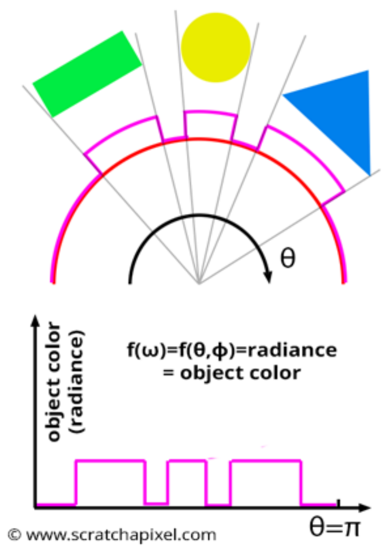


Figure 4: the amount of light reflected by objects in the scene towards P the shading point, can be written as function of solid angle, which we can also re-write in terms of spherical coordinates. In the 2D case, it is just a function of the angle θ. This is the function that we want to integrate over all directions contained in a half-disk in the 2D case and a hemisphere in 3D.

Ω (the capital Greek letter for omega) here represents the hemisphere of directions oriented about the normal at P. It defines the region of space over which the integral is performed.

What we integrate is generally a function. So what is that function in our case? If you look at figure 3, you can see the illumination of P by other objects in the scene, as a function. This function is non-zero in whatever parts of the half-disk onto which objects in the scene are projected to (assuming the objects themselves are not black). This function can be parametrised in different ways. It can be a function of solid angle (the Greek letter omega: ω) or a function of the spherical coordinates θ and ϕ (the Greek letters theta and phi).

These are just two different ways (but equally valid) of parametrising
the hemisphere. If we just consider the 2D case (figure 3), then this can simply be defined as function of the angle θ.

In this series of lessons on shading, we decided to avoid using terms from radiometry to make this introduction less technical. Though, it may be interesting at this point of the lesson to introduce the concept of **radiance** which you will probably come across if you read other documents on global illumination. Radiance is the term we should be using in this context to describe light reflected by objects in the scene towards P. It is often given the letter L.

If you are unfamiliar with the concept of integral, we really advice you to read the lessons from the Mathematics and Physics of Computer Graphics section. Though you can read or understand this sign as "collect the value that is to the right side of the integral sign (in this case L_i) over the region of space Ω (which again in this case, stands for the hemisphere oriented about N, the shaded point normal, or the half-disk in the 2D case)". Imagine for example that "rice grains" are actually spread across the surface of that hemisphere. The integral simply means "collect or count the number of grains that are spread over the surface of that hemisphere". Though note that "rice grains" are discrete elements. It is easy to count them but in our case, what we want is to measure the amount of light reflected by the surface of objects which are directly visible from P. This is a far more complex problem because these surfaces are continuous and can't be broken down into discrete elements like grains. But the principle stays the same.

You should start to see how we will solve the "which point on the surface of an object should we choose to compute the contribution of that object to the illumination of my shaded point P". The fact that we use an integral here and need to gather all light coming from all directions, suggests that it's not one point on the surface of that object that we need but all the points making up the surface of that objects. Of course trying to cover the surface of an object with points is not practical neither possible. Points are singularities, i.e. they have no area thus we can't actually represent a surface with an infinity of points which have no physical size. To solve the problem, one possible solution would be to divide the surface of the object into a very large number of very small patches and compute how much each patch making the surface of that object contributes to the illumination of P. This, in computer graphics, is technically possible and is called the **radiosity method**. A methods exists to solve the problem of light transfer between small patches and P but they have limitations (they are good for diffuse surfaces but not good for glossy or specular surfaces). More information on the radiosity method can be found in the next sections. Another method would consist of dividing the surface of the objects into small patches and simulating the contribution of these patches to the illumination of P by replacing them with small point light sources. This would somehow be possible and an advanced rendering technique called **VPL** is based on this exact principle, but in the end Monte-Carlo ray-tracing is just simpler and more generic.

> Monte Carlo is another way of solving the problem. It is essentially a **statistical method** that is based on the idea that you can **approximate** or **estimate** how much light is redirected towards P by other objects in the scene, by casting rays from P in random directions above the surface and evaluating the color of the objects these rays intersect (if they do intersect geometry). The contribution of each one of these rays is then summed up and the resulting sum is divided by the total number of rays. In pseudo-mathematical terms, you can write:
>
> $$\text{Gather Light} \approx \frac{1}{N} \sum_{n=0}^{N} \text{castRay}(\text{P}, \text{randomDirectonAboveP}) \,.$$
>
> This is quick note for readers who are already familiar with the concept of Monte Carlo integration. You probably already know that the complete equation to compute an approximation of an integral using Monte Carlo integration is:

$$\langle F^N \rangle = \frac{1}{N} \sum_{i=0}^{N-1} \frac{f(X_i)}{pdf(X_i)}.$$

If you are not familiar with Monte Carlo methods, you can find them explained in two lessons from the section Mathematics of Computer Graphics. If you look at the equation above, you will notice a PDF term which we will ignore for now, just for the sake of simplicity. We just want you to understand the principle of Monte Carlo integration at this point, so don't worry too much about the technical details which we will study in the next chapter.

The easiest way of making sense of the Monte Carlo integration method if you have never heard about it before, is to think of it in terms of polls. Let's say you want to know the average height of the entire adult population of a country. You can measure the height of each person, sum up all the heights and divide this sum by the total number of people. Though of course, in general there is too many people in this population and it would take too long to do so. We say the problem is intractable. What you do instead is measure the height of a small sample of that population, sum up the numbers and divide the sum by the sample size (let's call this number N). The technique works if people making up that sample are chosen **randomly** and with equiprobability (each individual making up the sample has the same chance to be part of the sample than any other individual making up the population). This is what we call a poll. The result that you get is probably not the exact number, but is generally a "good" estimation of that number. Increasing N, increases (in terms of probability) the quality of your estimation. The principle is exactly the same for approximating or estimating the amount of light reflected towards P by other objects in the scene. The problem is also intractable because we can't account for all the directions contained in the hemisphere (as mentioned before, surfaces are continuous and there's an infinity of points on their surface and therefore an infinity of directions, thus the problem is intractable here as well). Therefore we just select a few directions which we choose randomly and take the average of their contribution.

Which in pseudo-code translates to:

```
int N = 16; // number of random directions
Vec3f indirectLigthingApprox = 0;
for (int n = 0; n < N; ++n) {
    // cast a ray above P in random direction in hemisphere oriented around N, the surface n
    indirectLigthingApprox += castRay(P, randomDirectionAboveP); // accumulate results
}
// divide by total number of directions taken
indirectLigthingApprox /= N;
```

In CG, we call these random directions **samples**. What we actually do with Monte Carlo sampling is sampling or taking samples over the domain of the integral or the region of space over which the integral $\int_\Omega L_i$ is performed, which in this particular case is the hemisphere.

Now, don't worry yet about how we choose these random directions over the hemisphere and how many samples N we need to use. We will provide more information on this soon. For now what you need to remember from this part of the lesson, is that the Monte Carlo integration method only provides you with an **estimation** (or approximation) of the real solution which is the result of the integral $\int_\Omega L_i$. The "quality" of this approximation mostly depends on N, the number of samples used. The higher N, the more likely you are to get a result close to the actual result of this integral. We will come back on this topic in the next chapter.

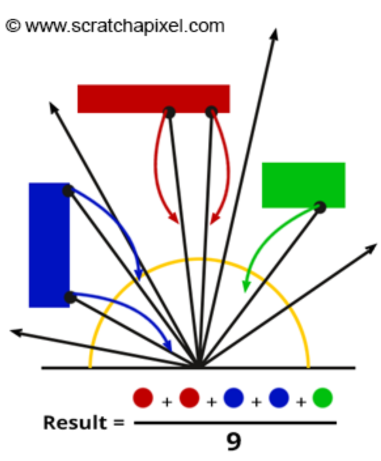


Figure 5: computing indirect illumination using the Monte Carlo integration method.

What do we mean by sampling? Since we need to account for the fact that light can come from anywhere above P, what we do instead in the case of Monte Carlo integration, is to select some random directions within the hemisphere oriented about P and trace rays in these directions into the scene. If these rays intersect some geometry in the scene, we then compute the object color at the intersected point which we assume to be the amount of light that the intersected object reflects towards P along the direction defined by the ray:

```
Vec3f indirectLight = 0;
int someNumberOfRays = 32;
for (int i = 0; i < someNumberOfRays; ++i) {
    Vec3f randomDirection = getRandomDirectionHemisphere(N);
    indirectLight += castRay(P, randomDirection );
}
indirectLight /= someNumberOfRays;
```

Remember that what we want, is to collect all light reflected towards P by objects in the scene, which we can write with the integral formulation: $\int_\Omega L_i$. The variable Ω here, represents the hemisphere of directions oriented around the normal at P. This is the equation we are trying to solve and to get an approximation of this integral using Monte Carlo integration, we need to "sample" the function on the right side of the integral sign (the $\int$ symbol), the L_i term.

> Sampling means take random directions from the domain of the integral we are trying to solve and evaluate the function on the right side of the integral sign at these random directions. In the case of indirect lighting, the domain of integral or the region of space over which the integral is performed, is the hemisphere of directions Ω.

Let's now see how this work with an example. For now we will work in 2D but later in the lesson we will provide the code to apply this technique to 3D.

Let's imaging that we want to compute the amount of light arriving indirectly at P. We first draw an ideal half disk above P and aligned along the surface normal at the shaded point. This half disk is just a visual representation of the set of direction over which we need to collect light reflected towards P by surfaces in the scene. Next, we will compute N random directions which you can see a N points located on the outer line of the half-disk whose locations on that line are randomly selected. In 2D, this can be done with the following code:

- step 1: what we do is first select a random location in the half disk of the unit circle which we can do by simply drawing a random value for the angle θ anywhere in the range $[0, \pi]$. Computing a random angle can simply be done using some C++ random function (such as drand48()) which, if it returns a random float in the range $[0, 1]$, needs to be properly remapped to the range $[0, \pi]$.

- step 2: next, notice that the "sampled" direction is located on a half disk which is not yet oriented along the shaded point normal. The best way of doing this is to first construct a 2x2 matrix using the shaded point normal and an interesting property of 2D vectors: a 2D cartesian coordinate system can be created from (V_x, V_y) and from $(V_y, -V_x)$. Both vectors define the up and right axis of a Cartesian coordinate system respectively. If you try to apply this technique to any 2D vectors, you will see that the vectors are always perpendicular to each other. Now that we have that matrix, we can use it to rotate our sample directions.
- step 3: we can finally trace a ray in the sampled direction. The goal of this step is to find out if the ray that we just sampled hits an object in the scene. If it does, we then compute the color at the point of intersection which we assume to be the amount of light the intersected object reflects towards P along the ray direction.
- step 4: now that we know how much light is reflected towards P by an object in the scene along the sample direction, we can treat the result of this intersection as if it was simply a light. In this particular example, we will assume that the object P belongs to, is diffuse. Thus, we will multiply the object color at the ray intersection point by the cosine of the angle between the surface normal N and the light direction which in our case, is the sample direction. We also assume that all other objects in the scene are diffuse (since we call the `castRay()` function recursively which on its own only compute the appearance of diffuse surfaces in this example).
- step 5: we accumulate the contribution of the sample.
- step 6: the Monte Carlo integration equation tells us we need to divide the sum of the samples contribution by the sample size, the number N.
- step 7: we can finally accumulate the diffuse color of the object at P due to direct and indirect illumination and multiply the sum of these two values by the object albedo (don't forget to divide the diffuse albedo by π, the normalising factor as explained in the introduction on shading).

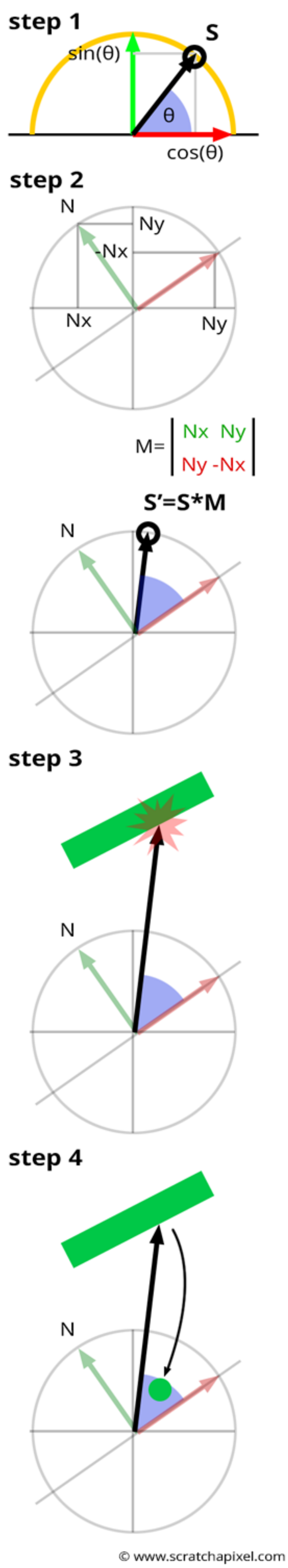


Figure 6: different steps to generate a sample. First generate

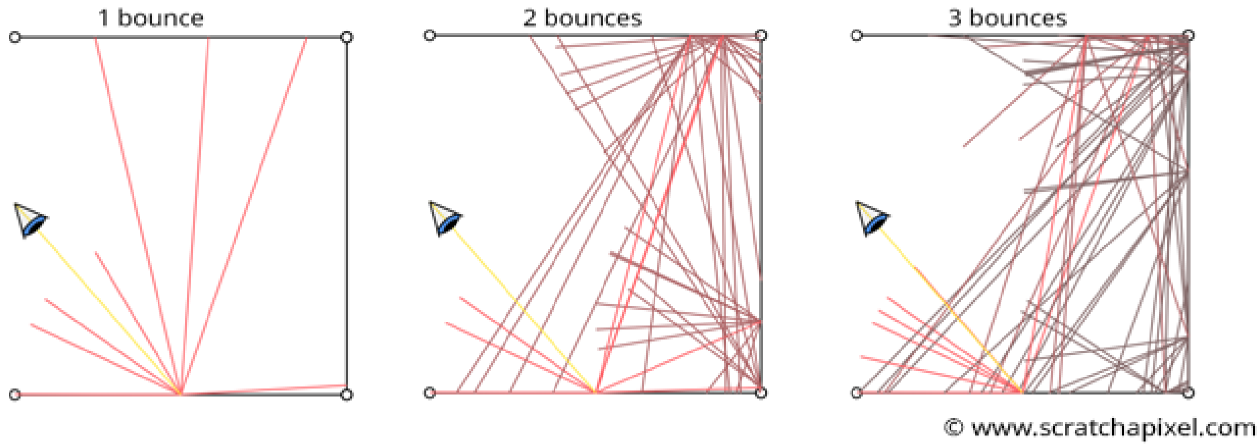
a sample in local Cartesian coordinate system and then transform this sample in a Cartesian coordinate system whose up vector is aligned with the shading normal. Finally, we can cast the ray into the scene. If it intersects an object, we compute the color of the object at the intersection point. The color of all the objects intersected are added up, and the resulting sum is divided by the number N (the number of generated samples).

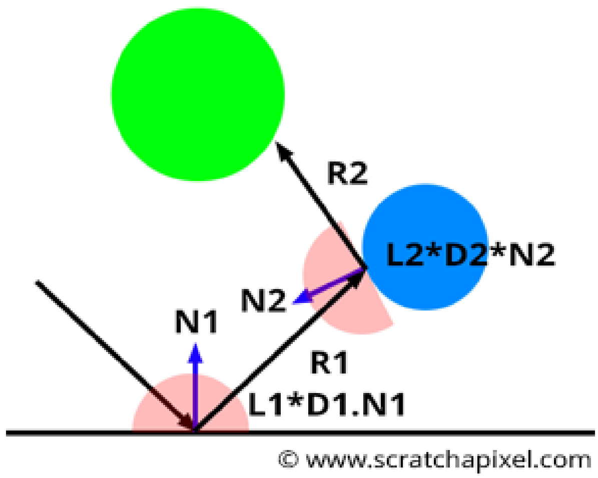
```
vec3f castRay(Vec2f P, Vec2f N, uin32_t depth) {
    if (depth > scene->options.maxDepth) return 0;
    Vec2f N = ..., P = ...; // normal and position of the shaded point
    Vec3f directLightContrib = 0, indirectLightContrib = 0;
    // compute direct illumination
    for (uint32_t i = 0; i < scene->nlights; ++i) {
        Vec2f L = scene->lights[i].getLightDirection(P);
        Vec3f L_i = scene->lights[i]->intensity * scene->lights[i]->color;
        // we assume the surface at P is diffuse
        directLightContrib += shadowRay(P, -L) * std::max(0.f, N.dotProduct(L)) * L_i;
    }
    // compute indirect illumination
    float rotMat[2][2] = {{N.y, -N.x}, {N.x, N.y}}; // compute a rotation matrix
    uint32_t N = 16;
    for (uint32_t n = 0; n < N; ++n) {
        // step 1: draw a random sample in the half-disk
        float theta = drand48() * M_PI;
        float cosTheta = cos(theta);
        float sinTheta = sin(theta);
        // step 2: rotate the sample direction
        float sx = cosTheta  * rotMat[0][0] + sinTheta  * rotMat[0][1];
        float sy = cosTheta  * rotMat[1][0] + sinTheta  * rotMat[1][1];
        // step 3: cast the ray into the scene
        Vec3f sampleColor = castRay(P, Vec2f(sx, sy), depth + 1); // trace a ray from P in ran
        // step 4 and 5: treat the return color as if it was a light (we assume our shaded sur
// diffuse shading = L_i * cos(N.L)
        IndirectLightContrib += sampleColor * cosTheta;
    }
    // step 6: divide the result of indirectLightContrib by the number of samples N (Monte Car
    indirectLightContrib /= N;

    // final result is diffuse from direct and indirect lighting multiplied by the object colo
    return (indirectLightContrib + directLightContrib) * objectAlbedo / M_PI;
}
```

If you are an expert in Monte Carlo methods, you probably noticed that we omitted to divide the sample contribution by the random variable PDF. We have omitted this part by choice to avoid confusing readers who don't know anything about the concept of PDF. Check the next chapter for a complete implementation.

Et voila! You are done. You might start to see why the technique is expensive though. Basically when you want to compute indirect illumination (**indirect diffuse** in this case as we simulate how diffuse surfaces illuminate each other), you need to shoot many rays (16 in the example above), and every time one of these rays intersects a diffuse surface, you repeat the same process again: the process is recursive. After 3 bounces, this is already 16 * 16 * 16 = 4096 rays cast into the scene (assuming all rays intersect geometry). This number increases exponentially as the number of bounces increases.

For this reason, as in the case of mirror reflections, we stop casting rays after a certain number of bounces (for example 3) which in the code above is controlled by the `maxDepth` variable once again.





Note that the contribution of a bounce decreases as the ray depth increases. This because each time a new ray is cast from a surface its contribution is attenuated by the cosine law. If we just consider 2 bounces for now, a ray is cast in the scene and intersects an object in P_1 (bounce 1). The color at this intersection point C_1 is attenuated by the dot product between the shading normal and the ray direction D_1. Another ray is cast from P_1 and intersects an object in P_2. We then attenuate the contribution of this ray R_2 by the cosine of the angle between the shading normal at P_2 and the ray direction R_2 but the contribution of this ray is itself attenuated again later on by the cosine of the angle between the shading normal at P_1 and the first sample direction D_1. As you can see the contribution of the second bounce is attenuated twice. The contribution of a third bounce would be attenuated three times, etc. As a result, there is a point where simulating more bounces makes very little difference to the final image (this depends on the scene configuration but for diffuse surfaces, this is generally the case after 4 or 5 bounces).

Technically, you know how to simulate indirect and indirect lighting which should allow you to create very realistic pictures. Though the problem is not that simple. First in this example, all we have learned to do is to simulate indirect lighting by diffuse surfaces. First, perfect diffuse surfaces almost never exist in nature. But more importantly, most scenes contain a combination of diffuse and specular materials. We didn't even speak yet of other kind of materials such as translucent materials which are not exactly diffuse. Now you may think, why wouldn't Monte Carlo integration work to simulate over kind of materials? In fact it does work but the method reveals itself to be inefficient when the scene contains a mix of specular or transparent and diffuse surfaces. Let's see a few examples.

Using Indirect Diffuse Sampling Method to Compute Indirect Specular Is Inefficient

As you already know, there is essentially two kinds of material: specular and diffuse. As explained in the previous lesson, all light sources (whether direct or indirect) which are located above a point located on the surface of a diffuse object contribute to the brightness of that point. Of course the contribution of each light source (again whether direct or indirect) is somehow attenuated by the cosine of the angle between the light direction and the surface normal but we will ignore this detail for now. Remember that a diffuse reflection is view independent. A diffuse surface will always have the same brightness regardless of which angle you look at it. Specular surfaces are different: they are view dependent. you can only see the reflection of an object if the angle of view is roughly aligned with the mirror reflection along which the light of this object is reflected. This essentially means that light rays reflected by a glossy surface are reflected within a cone of directions oriented about the reflection direction, as explained in the previous chapter. Remember that when we compute the contribution of an object to the illumination of a diffuse object, we speak of indirect diffuse. Similarly, when we want to compute the contribution of an object to the illumination of a glossy object, we speak of **indirect specular**. Remember two additional things:

- For a glossy surface, a beam of light with a given incident direction, is reflected in a cone of directions (the specular lobe). How wide is that cone depends on the surface roughness. The rougher the surface, the larger the cone.
- Reflection by surfaces is a **reciprocal process**: swapping the view and the incident direction does not change the way a surface reflects light.

For a diffuse surface, a light always contributes to the illumination of P regardless of its position (as long as the light is above the surface). For a specular surface though, a light can only be visible if the view direction V is somehow roughly aligned with the mirror direction R of the incident light direction. How close V and R have to be, depends on the surface roughness. If the surface is very rough, some light might still be reflected by the surface along the view direction even if V is not closely aligned with the mirror direction R as show in figure 7 (top). Though as soon as V is totally outside the cone of directions in which the specular surface reflects light rays into, then of course in this case, there should be no light reflected at all (figure 7 - top). We can use this observation to compute the glossy reflection of an object by a glossy surface. If we know the view direction V then only light rays with certain incident directions will be reflected towards the eyes along V. Now imagine you compute the mirror direction of the vector V as shown in figure 7 (bottom). This gives us the vector R_V. Let's now imagine the cone of directions in which light rays are reflected but centred around V_R. What happens? Well, this tells us that any light ray whose incident direction is contained within that cone of direction will somehow be reflected along V to at least some quantity greater than 0.

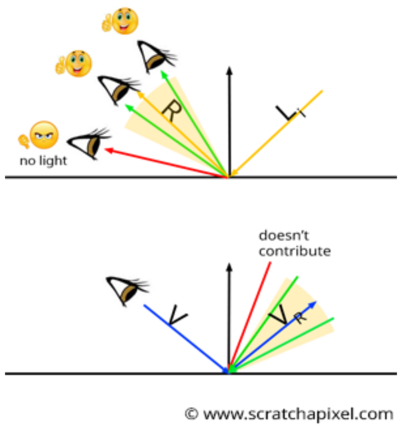
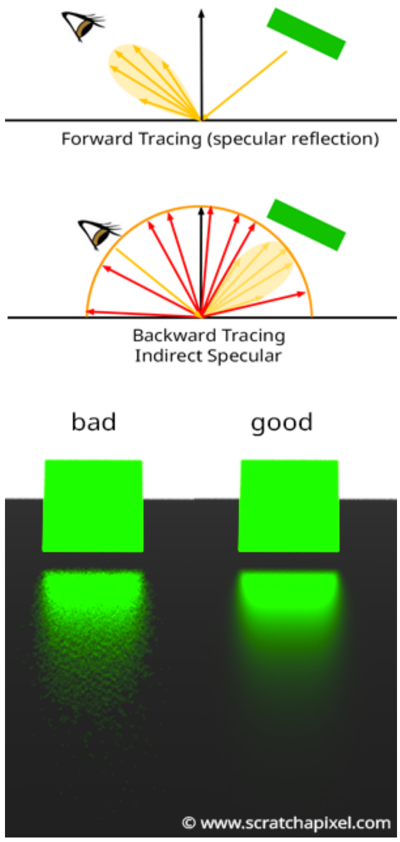


Figure 7: rays contained within the cone are reflected towards the eye. This is a consequence of the reciprocity rule.

> This is a direct consequence of the reciprocity rule: all light reflected along V has to have an incident direction contained within the cone of directions centred around V_R .

Here is why the technique we used to compute indirect diffuse doesn't work well for indirect specular. We have just explained that only light rays contained within the cone of direction centred around V_R somehow contribute to the amount of light that is reflected along V. Thus technically we should only be interested in directions contained within that cone (as shown in figure 7). The problem though with the technique we used so far, which consists of sampling the entire hemisphere, is that many of the created directions are not contained within that cone at all. Another way of saying this is that because the direction of the samples is chosen randomly, there is no guarantee that any of the rays will actually be contained within the specular lobe's cone of directions at all (figure 8



- middle). Even if just a few rays are contained in this cone, all the rays which are not contained within the cone itself are somehow wasted anyway since we know than even if they hit an object, their contribution will be null. So the technique is essentially inefficient because many of the rays we cast if we sample the hemisphere are potentially wasted.

Figure 8: hemispherical sampling is not efficient to sample a specular lobe. Many of the cast rays are not contained within the specular lobe's cone of directions.

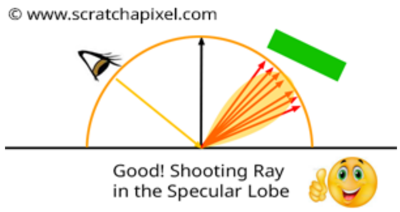


Figure 9: an idea sampling method for glossy objects would be to generate all samples within the specular lobe's cone of directions.

Obviously this is a very quick and simplistic view of what the problem is. If you are already familiar with Monte Carlo integration then you will understand that ideally you want all the samples that you use to somehow sample the function that you try to integrate in places where the function is not zero. Doing otherwise will increase what we call **variance** which visually translates into what we call **noise**. You can see the difference in the render of the reflection of the green square by a glossy surface in figure 8. On the left, we can see an example of an image which has a lot of a noise compared to the image on the right which seems almost noiseless. Though the two images were rendered with the same number of samples N. On the left, we have high variance and on the right low-variance. A full lesson in the next section will be devoted to explain how this can be done. Many techniques can be used to reduce variance (variance reduction), such as for example importance sampling. You can find more information on noise and the relationship between N, the number of samples used and the amount of noise in the image, in the next chapter.

Your intuition should tell you though that the solution to the problem we have just described is to cast rays such they are they are all contained with the specular lobe's cone of directions (figure 9). Their direction should still be random because this is a requirement of the Monte Carlo integration method, but we should cast them in directions which are useful. They are different ways of doing so, but we will study them in a lesson on advanced global illumination which you can find in the next section.

Caustics: The Nightmare of Backward Tracing

We already talked about the difficulty of simulating caustics in the introduction to the topic of light transport which you can find in the lesson Rendering an Image of a 3D Scene: an Overview. Caustics are the results of light rays being focused in specific points or regions of space by a specular surface or as a result of refraction (light rays passing through a glass of water or wine are focused to a point when then exist the glass as shown in figure 10 and 11.

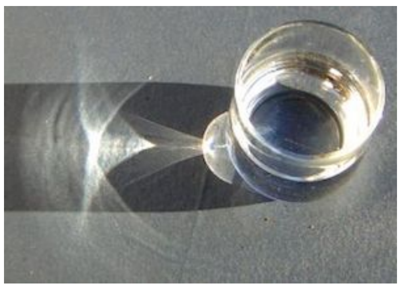


Figure 10: caustics.

The problem with caustics is similar to the problem we had previously with distributing samples over the hemisphere to simulate indirect specular reflections. Many of the rays as you can see in figure 11 do not intersect with the sphere which though concentrates all light rays towards P the shaded point. One ray intersects the sphere but it intersects it on a point far from the region where light rays are gathered and leave the sphere. Even though P is likely to be very bright as a result of many light rays converging to its location, our Monte Carlo method totally fails finding out where in the scene light illuminating P comes from. As a result, the shaded point will appear dark in the render when in reality it should be quite bright. Simulating caustics that way creates highly noisy images as shown in figure 12.

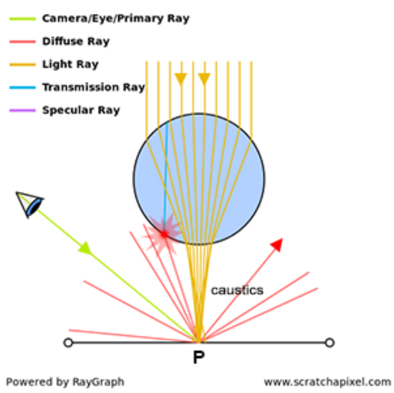


Figure 11: hemispherical sampling is not a good method to simulate

Caustics are different than indirect specular reflections, in the sense that at least in the case of indirect reflections we have some prior knowledge of the directions in which we should cast the rays. We said they should be contained with the specular lobe's cone of directions and we explained why. Though in the case of caustics, we don't know where in the scene these caustics are created. We don't know which parts of the scene concentrate and redirect light rays towards P. Of course, we can increase the number of samples N, but you may need to increase it to a very large number in order to reduce variance significantly, and this comes at the price of increasing render time greatly. This is clearly not a good solution. A good solution is one in which we decrease variance significantly while keeping render times low.

caustics. In this particular case many rays miss the sphere, and the few that hit the sphere do not hit the surface of the sphere where they rays are being focused by the glass ball.

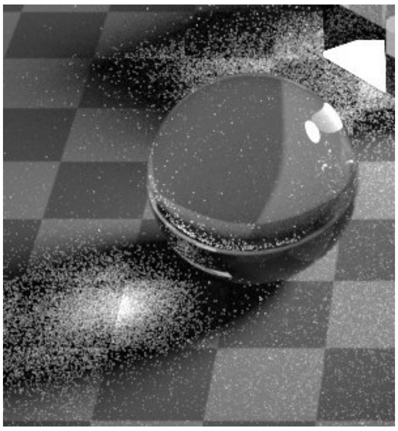


Figure 12: backward tracing is not a good solution for simulating caustics. It generates very noisy images as it fails to efficiently find where light is being focused by refractive or very reflective objects (light sources are not directly visible).

In reality there is not really good solutions to simulate caustics using backward tracing which is why researcher came up with other approaches. One of the most common one is to actually simulate the propagation of light by highly specular and transparent objects from the scene using forward tracing in a pre-pass. When light rays after interacting with glossy or refractive materials hit diffuse surfaces, their contribution to the point where they hit the diffuse object is stored in a spatial data structure. At render time (after the pre-pass, when we compute the final image), when a diffuse surface is being hit by a primary ray for instance, we look into this spatial data structure whether some energy was deposited there during the pre-pass by some specular or transparent objects. This is the foundation of a technique called **photon mapping** which we will talk about in another section.

More generally the idea of combining both backward and forward tracing to solve complex light transport provides better results than just using backward or forward tracing alone (in the photon mapping example, we use forward tracing in the pre-pass to follow the path of light rays as they interact with glossy and refractive objects and backward tracing to solve for visibility and direct lighting). This is the approach used by many light transport algorithm such as photo mapping which we already talked about or **bidirectional path tracing** which we will study in the next section. Of course these other light transport algorithms have their own limitations.

What's Next?

We will study and implement in our renderer indirect diffuse using Monte Carlo integration.