Global Illumination and Path Tracing

Contents

Global Illumination in Practice: Monte Carlo Path Tracing

In the previous chapter, we introduced the principle of Monte Carlo integration to simulate global illumination. We also explained that uniformly sampling the hemisphere works well in practice for simulating indirect diffuse, but not well for simulating indirect specular or caustics. For this reason, in this lesson, we will limit ourselves to simulating indirect diffuse: that is, how diffuse objects illuminate each other as the "indirect" effect of these objects reflecting light themselves and acting somehow as a source of light.

We showed in the first chapter, how the Monte Carlo integration method worked in 2D. Extending the method to 3D is not "simple" and works on the same principle. Here are the steps (which as you can see are very similar to the steps we followed in the 2D case).

Concept: ideally we want to create samples on the hemisphere oriented about the normal N of the shaded point P. Though as in the 2D case, it is easier to create these samples in a hemisphere oriented with the y-axis of some "canonical" Cartesian coordinate systems and then transform then into the shaded point local coordinate system (in which the hemisphere is oriented along the normal as shown in figure 1 - bottom). The reason why we prefer to create samples in a hemisphere whose up vector is oriented with the y-axis of the world Cartesian coordinate system, is because the coordinates of the sample in this coordinate system can be easily computed with the basic spherical-to-Cartesian coordinates equations, which we know are:

$$
\begin{aligned}
x &= \sin\theta\cos\phi, \\
y &= \cos\theta, \\
z &= \sin\theta\sin\phi,
\end{aligned}
$$

and then transformed in the shaded point local coordinate system. We will explained in this lesson how samples are created, how to create a new coordinate system in which the shading normal N defines the up axis of that coordinate system, and how the other axis of that local Cartesian coordinate system (which are technically tangent to the surface at P and perpendicular to N) are also computed. If you read the lesson on Geometry, you should know that once we have the axes of an orthogonal coordinate system C, we can transform any point or vector (aka direction) from any Cartesian coordinate system to C.

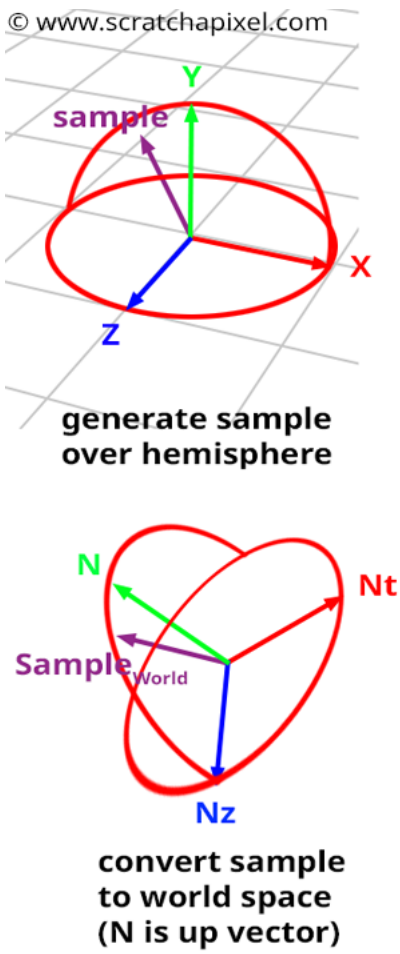


Figure 1: transforming sampling from the coordinate system in which they were created to world space (in the shaded point local coordinate system whose up vector is aligned with N).

- Step 1: create a Cartesian coordinate system in which the up vector is oriented along the shaded point normal N (the shaded point normal N and the up vector of the coordinate system are aligned).

- Step2: create a sample using the spherical to Cartesian coordinates equations. We will show in this chapter how this can be done in practice.
- Step 3: transform the sample direction from the original coordinate system to the shaded point coordinate system.
- Step 4: trace a ray in the scene in the sampled direction.
- Step 5: if the ray intersects an object, compute the color of that object at the intersection point and add this result to a temporary variable. Because the surface is diffuse, don't forget to multiply the light intensity returned along each ray by the dot product between the ray direction (the light direction) and the shaded normal N (see below).
- Step 6: repeat step 2 to 5 N-times.
- Step 7: divide the temporary variables that holds all the results of all the sampled rays by N, the total number of samples used. The final value is an approximation of the shaded point indirect diffuse illumination. The illumination of P by other diffuse surfaces in the scene.

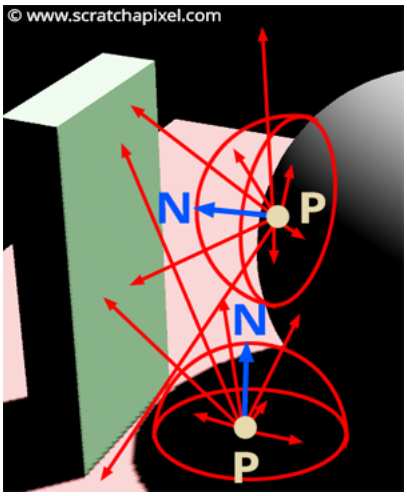


Figure 2: the principle of Monte Carlo integration applied to a 3D scene. Rays are randomly generated into a hemisphere of directions oriented around the shading normal N. If these rays hit an object, the color at the intersection point is computed and returned to the shading point. All the colors are summed up and divided by the number N of cast samples. Monte Carlo theory also implies to divide each sample by the random variable PDF. We shouldn't also forget to multiply the object's color by the cosine of the angle between the shading normal and the ray direction (which in this case can be seen the light direction).

Technically, we need to divide each sample contribution by the random sample PDF. This will be explained later in this chapter.

In pseudo-code we get:

```
001 Vec3f castRay(Vec3f &orig, Vec3f &dir, const uint32_t &depth, ...)
002 {
003     if (depth > options.maxDepth) return 0;
004     Vec3f hitPointColor = 0;
005     // compute direct ligthing
006     ...
007     // step1: compute shaded point coordinate system using normal N.
008     ...
009     // number of samples N
010     uint32_t N = 16;
011     Vec3f indirectDiffuse = 0;
012     for (uint32_t i = 0; i < N; ++i) {
013         // step 2: create sample in world space
014         Vec3f sample = ...;
015         // step 3: transform sample from world space to shaded point local coordinate s
016         sampleWorld = ...;
017         // step 4 & 5: cast a ray in this direction
018         indirectDiffuse += N.dotProduct(sampleWorld) * castRay(P, sampleWorld,
```

```
019          depth + 1, ...);
020      }
021      // step 7: divide the sum by the total number of samples N
022      hitPointColor += (indirectDiffuse / N) * albedo;
023      ...
024      return hitPointColor;
}
```

> What the `castRay` does, is essentially returning the amount of light that an object reflects towards the shaded point along the direction `sampleWorld` (if the ray intersected an object). The principle here to compute the contribution of that light ray to the color of P is exactly the same than with standard light sources. In other words, if the surface is diffuse, we should also multiply the light intensity by the cosine of the angle between the light direction (`sampleWorld`) and the shading normal N (line 18).

Let's see how to implement these steps one by one.

Sep 1: Create a Local Coordinate System Oriented along the Shading Normal N

We want to build a Cartesian coordinate system in which the normal N is aligned with the up vector (or y-axis) of that coordinate system. What we know from the lesson on Geometry, is that a vector can also be interpreted as the coefficients of a plan equation and that the plane in question is perpendicular to that vector:

$$A * x + B * y + C * z = D = 0,$$
$$N_x * x + N_y * y + N_z * z = D = 0.$$

In our case, we don't care about the variable D which set to 0 because, since we deal with vectors in this case, the plane passes through the world origin. Since we are looking to build a Cartesian coordinate system, we can assume that any vector lying on that plane is perpendicular to N. Let's call this vector N_T. We could use N_T as the second axis of our coordinate system. Now by taking the cross product of N_T and N we would create a third vector N_B perpendicular to both N_T and N. Et voila! N, N_T and N_B form a Cartesian coordinate system. So how do we find a vector lying in the plane anyway?

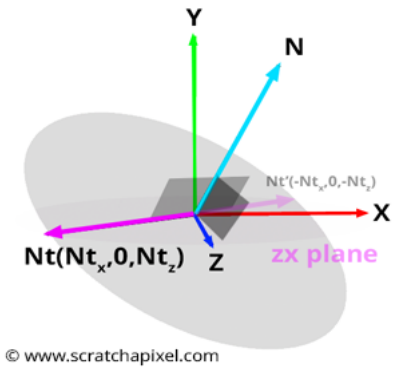


Figure 3: two vectors lying in the plane perpendicular to the normal N exist.

If you look at figure 3, you will realise the one of the vectors lying in the plane has actually its y-coordinate equals to 0. So we can assume that there is one such vector lying in this plane whose y-coordinate is actually 0 which reduces our plane equation to two unknowns:

$$N_x * x + N_y * (y = 0) + N_z * z = N_x * x + N_z * z = 0.$$

We can re-write this equation as follows:

$$N_x * x = -N_z * z.$$

Note that this equation is true if $x = N_z$ and $z = -N_x$ or if $x = -N_z$ and $z = -N_x$. So this is simple, if we take either $N_T = N_z, 0, -N_x$ or $N_T = -N_z, 0, N_x$ then we have a vector lying in the plane perpendicular to N. You can choose either option as long as you take care of creating a right hand coordinate system which also depends on the order in which you order N and N_T in the final cross product. The only thing we need to take care of, is to normalise N_T:

```
001 void createCoordinateSystem(const Vec3f &N, Vec3f &Nt, Vec3f &Nb)
002 {
003     Nt = Vec3f(N.z, 0, -N.x) / sqrtf(N.x * N.x + N.z * N.z);
004     ...
005 }
```

All there is left now is to compute N_B with a cross product:

```
001 void createCoordinateSystem(const Vec3f &N, Vec3f &Nt, Vec3f &Nb)
002 {
003     Nt = Vec3f(N.z, 0, -N.x) / sqrtf(N.x * N.x + N.z * N.z);
004     Nb = N.crossProduct(Nt);
005 }
```

This construction though is not the best if the normal vector's y-coordinate is greater than its x-coordinate. In this particular case, it is best for numerical robustness to build a vector within the plane whose x-coordinate is equal to 0. If you watch figure 3 again, you will see that this also an option:

$$N_x * (x = 0) + N_y * y + N_z * z = N_y * y + N_z * z = 0.$$

The same logic applies here to find a possible solution for y and z. Finally we get:

```
001 void createCoordinateSystem(const Vec3f &N, Vec3f &Nt, Vec3f &Nb)
002 {
003     if (std::fabs(N.x) > std::fabs(N.y))
004         Nt = Vec3f(N.z, 0, -N.x) / sqrtf(N.x * N.x + N.z * N.z);
005     else
006         Nt = Vec3f(0, -N.z, N.y) / sqrtf(N.y * N.y + N.z * N.z);
007     Nb = N.crossProduct(Nt);
008 }
```

Our local coordinate system is built! Now all we have to do is learn how to create samples, and then transform these samples in this coordinate system.

Create Samples on the Hemisphere

As you know, we can define the position of a point on the sphere (or hemisphere) using either polar coordinates or Cartesian coordinates. In polar coordinates a point or a vector (we are interested in direction here), are defined with two angles: θ (the greek letter theta) and ϕ (the greep letter phi). The first angle is contained in the range $[0, \pi/2]$ and the second angle (ϕ) is contained in the range $[0, 2\pi]$. Remember that the Monte Carlo integration method works if we chose the direction of the samples randomly. Now this is where Monte Carlo theory kicks in. Remember that what we want to do here is uniformly sample the hemisphere with respect to solid angle. Now we won't explain the theory of Monte Carlo again this section. If you haven't read the lesson on Monte Carlo yet, now is the time to do so (Mathematical Foundations of Monte Carlo Methods and Monte Carlo in Practive). Anyway, be ready: we are going to have to deal with some maths now.

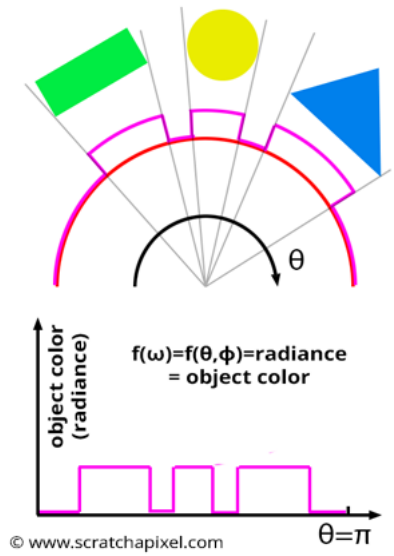


Figure 4: the amount of light reflected by objects in the scene towards P the shading point, can be written as function of solid angle, which we can also re-write in terms of spherical coordinates. In the 2D case, it is just a function of the angle θ. This is the function

Now the actual equation (which is called an **estimator**) that we are going to be using looks as follows (read "the Monte Carlo estimator for the integral of the function f(x), with probability density function pdf(x) is"):

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

that we want to integrate over all directions contained in a half-disk in the 2D case and a hemisphere in 3D.

This is the generic form of a Monte Carlo estimator which we can use to approximate the result of an integral. What we want to integrate in this particular case is the function $f(x)$ and as explained in the previous chapter, in our case this function can be seen as the amount of light arriving at P the shaded point from every direction in the hemisphere oriented around the normal at P. It is maybe simpler to imagine what this function can look like if you consider the 2D case (or maybe just a slice though the hemisphere) as shown in figure 4. Now as usual, the idea here is to sample that function for random directions. In the equation above, this is represented by the random variable X_i. So the idea is that we generate a random direction D_R in the hemisphere and evaluate the function for that direction, which in our case means "cast the ray from P in the direction D_R into the scene and return the color of the object that this ray intersected (if it intersected an object at all)". That color represents the amount of light that the object the ray intersected reflects towards P along the ray direction D_R. Mathematically we also need to **divide** this result by the probability of generating the direction D_R. This is represented in the equation above by the term called PDF (this stands for probability distribution function). The result of this function could be unique for each random direction X_i. Which is why it is defined as a function. Hopefully, because in our case we want to generate uniformly distributed samples (or directions), all the rays will have the same PDF. In other words, the PDF in this case is constant, because all rays or all directions have the same probability of being generated (we speak of equiprobability). So let's find what this PDF is and let's then learn how to generate these random uniformly distributed samples or directions.

To sample a function we need to first find the PDF of that function, then compute its CDF then finally the inverse of that CDF. These steps were explained in detail in the lesson Mathematical Foundations of Monte Carlo Methods. First, keep in mind that we want to uniformly sample the hemisphere. That means that any of the sample or direction we will draw should have the same probability than any other drawn directions. The probably to draw direction A is the same than the probability to draw direction B, C, D, etc. Now, we know that the probability of drawing a sample from a given function is given by the function's PDF (probability distribution function). We also know that all PDFs must integrate to 1 over their sample space (or domain of integration). In the case of the hemisphere, we will first define that sample space in terms of solid angle: the solid angle of a hemisphere is equal to 2π. And since all samples have the same chance (aka probability) of being generated, then our PDF must also be constant. If we write our PDF as $p(x)$ and then integrate over the entire sample space (2π), then we get (where again we will express values in terms of solid angle to start with):

$$\int_{\omega=0}^{2\pi} p(\omega)\, d\omega = 1.$$

Because, as mentioned before, a PDF must integrate to 1 over the function sample space. Since $p(\omega)$ is a constant, we can write (we replace the PDF by the constant C in the following equation):

$$C \int_{\omega=0}^{2\pi} d\omega = 1.$$

We know (check the lesson the Mathematic of Shading in the section Mathematics of Computer Graphics) that:

$$\int_{x=a}^{b} dx = b - a.$$

Thus:

$$C * \int_{\omega=0}^{2\pi} d\omega = C * (2\pi - 0) = 1.$$

From which we can find C or the PDF of our function:

$$C = p(\omega) = \frac{1}{2\pi}.$$

So far, so good. Now it's great to know the PDF of our function we want to sample to start with, but the problem is that we can't really work with solid angles. Our ultimate goal is to generate random directions contained within the hemisphere so we need to express this PDF in terms of polar coordinates i.e. in terms of θ and ϕ:

Now we know that the probability that a given solid angle belongs to a larger set can be written as:

$$Pr(\omega \in \Omega) = \int_{\Omega} p(\omega)\, d\omega.$$

This is just a generalisation of the equation we integrated above, but in the previous equation, ω was equal to $2\pi+$, in other words ω covered the area of the entire hemisphere. Now, if want to express the same probability density function but this time in terms of θ and ϕ, we can write:

$$p(\theta, \phi)\, d\theta\, d\phi = p(\omega)\, d\omega.$$

We learned in the previous lesson that a differential solid angle can be re-written in terms of polar coordinates as follows:

$$d\omega = \sin\theta\, d\theta\, d\phi.$$

If we substitute this equation in the previous equation we get:

$$p(\theta, \phi)\, d\theta\, d\phi = p(\omega)\, d\omega,$$
$$p(\theta, \phi)\, d\theta\, d\phi = p(\omega)\, \sin\theta\, d\theta\, d\phi.$$

The $d\theta$ and $d\phi$ on each side of the equation cancel out and we get:

$$p(\theta, \phi) = p(\omega)\, \sin\theta,$$
$$p(\theta, \phi) = \frac{\sin\theta}{2\pi}.$$

We just replaced $p(\theta)$ with the value we computed above: $p(\omega) = 1/2\pi$. A little more patience, we are almost there. The function $p(\theta, \phi)$ is called a join probability distribution. Now, we need to sample θ and ϕ independently. In this particular case, the PDF is said to be separable. In other words, we can split the PDF in two PDFs, one for θ and one for ϕ but we can also use another technique, which relies on the concept of marginal and conditional PDF. A marginal distribution is one in which the probability of a given variable (for example θ) can be expressed without any reference to the other variables (for example ϕ). You can do this by integrating the given PDF with respect to one of the variables. In our case, we get (will integrate $p(\theta, \phi)$ with respect to ϕ):

$$p(\theta) = \int_{\phi=0}^{2\pi} p(\theta, \phi)\, d\phi = \int_{\phi=0}^{2\pi} \frac{\sin\theta}{2\pi}\, d\phi = 2\pi * \frac{\sin\theta}{2\pi} = \sin\theta.$$

The conditional probability function gives the probably function of one of the variables (for example ϕ) when the other variable (θ) is known. For continuous function (which the case of our PDFs in this

example) this can simply be computed as the joint probability distribution $p(\theta, \phi)$ over the marginal distribution $p(\theta)$:

$$p(\phi) = \frac{p(\theta, \phi)}{p(\theta)} = \frac{1}{2\pi}.$$

Now that we have the PDFs for θ and ϕ, we need to compute their respective CDFs and inverse them. Remember from the lesson on Monte Carlo that a CDF can be seen as a function that returns the probability that some random variable X will be found to have a value less or equal to x. To take a practical example, let's say "if I randomly generate a random value for θ, what is the probability that θ takes on a value less than or equal to say for example $\pi/4$? We can compute this CDF as an integral of the PDF as follows:

$$Pr(X <= \theta) = P(\theta) = \int_0^\theta p(\theta)\, d\theta = \int_0^\theta \sin\theta\, d\theta.$$

The **Fundamental Theorem of Calculus** says that if $f(x)$ is continuous on [a, b] and if If $F(x)$ is any antiderivative of $f(x)$, then

$$\int_a^b f(x)\, dx = F(b) - F(a).$$

The antiderivative of $\sin x$ is $-\cos x$, thus:

$$\int_0^\theta \sin\theta\, d\theta = [-\cos\theta]_0^\theta = [-\cos\theta - -\cos 0] = 1 - \cos\theta.$$

The CDF of the $p(\phi)$ is simpler:

$$Pr(X <= \phi) = P(\phi) = \int_0^\phi \frac{1}{2\pi}\, d\phi = \frac{1}{2\pi}[\phi - 0] = \frac{\phi}{2\pi}.$$

Let's finally invert these CDFs:

$$y = 1 - \cos\theta,$$
$$\cos\theta = 1 - y,$$
$$\theta = cos^{-1}(1 - y).$$

$$y = \frac{\phi}{2\pi},$$
$$\phi = y * 2\pi.$$

Congratulation. The rest of the Monte Carlo theory now tells us that if we draw some random variable uniformly distributed in the range [0,1], and replace these random numbers in the equations above, then we get uniformly distributed values of θ and ϕ which we can then use in the polar-to-Cartesian coordinates equations to compute a random uniformly distributed direction over the hemisphere. Let's see how this works in practice:

```
001 Vec3f uniformSampleHemisphere(const float &r1, const float &r2)
002 {
003     // cos(theta) = r1 = y
004     // cos^2(theta) + sin^2(theta) = 1 -> sin(theta) = srtf(1 - cos^2(theta))
005     float sinTheta = sqrtf(1 - r1 * r1);
006     float phi = 2 * M_PI * r2;
007     float x = sinTheta * cosf(phi);
```

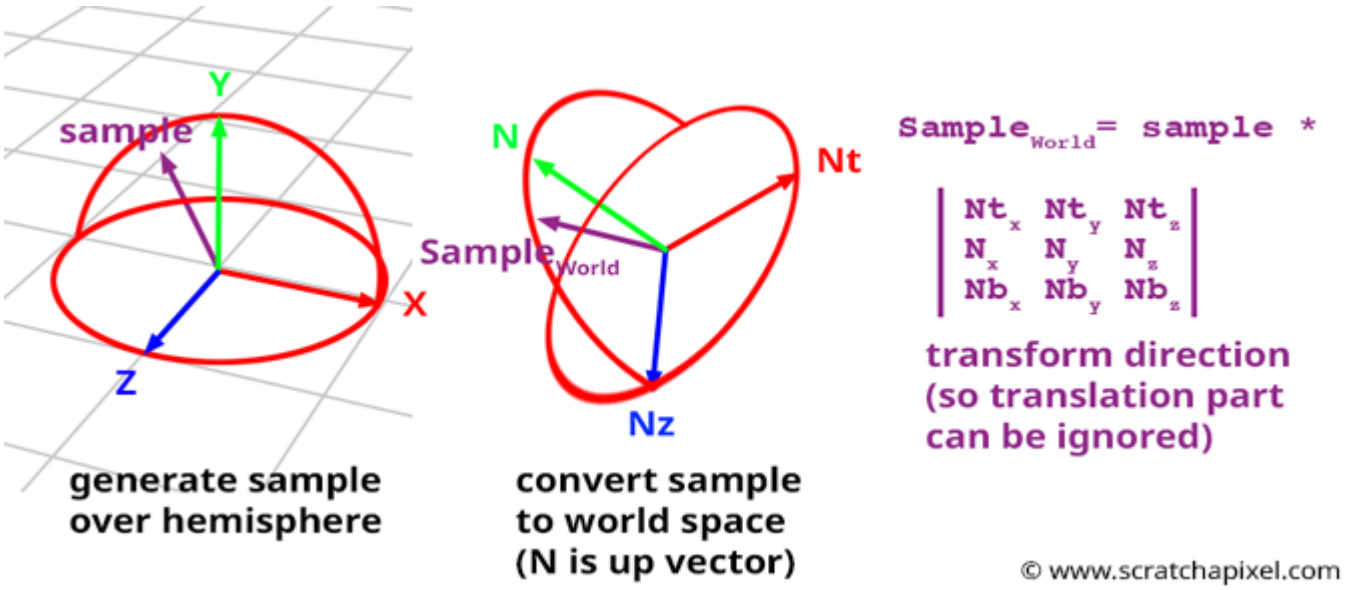
```
008     float z = sinTheta * sinf(phi);
009     return Vec3f(x, u1, z);
010 }
011
012 ...
013 std::default_random_engine generator;
014 std::uniform_real_distribution<float> distribution(0, 1);
015
016 uint32_t N = 16;
017
018 for (uint32_t n = 0; n < N; ++i) {
019     float r1 = distribution(generator);
020     float r2 = distribution(generator);
021     Vec3f sample = uniformSampleHemisphere(r1, r2);
022     ...
023 }
```

There is a couple of interesting things to note about the code above. Note that when we inverted the first CDF one of the equations was $\cos\theta = 1 - y$. The y variable is in fact the random variable uniformly distributed between 0 and 1 that we want to use in order to randomly pick a value for θ. So $1 - y$ or $1 - \epsilon$ if ϵ, the Greek letter epsilon which represents our uniformly distributed random variable, is equal to $\cos\theta$. First $\cos\theta$ is the y-coordinate of the random direction. So we can set the y-coordinate of our random direction with $1 - \epsilon$ directly (line 9). Then note that $\cos^2\theta + \sin^2\theta = 1$ thus $\sin\theta = \sqrt{1 - cos^2\theta}$. If we replace $\cos\theta$ with $1 - \epsilon$ again, then we get $\sin\theta = \sqrt{1 - (1 - \epsilon) * (1 - \epsilon)}$. Then finally note that because our random variable ϵ is uniformly distributed actually using $1 - \epsilon$ or ϵ is the same. It doesn't matter since there is as much probability to draw a $1 - \epsilon$ than a ϵ and since both number have the same probability to be generated, one can be used in place of the other (and vice versa). This simplifies our equation to $\cos\theta = \epsilon$ and $\sin\theta = \sqrt{1 - \epsilon * \epsilon}$.

Great, it was a lot of maths, but now you know how to generate uniformly distributed random samples in the hemisphere. Let's move to the next step and transform this sample to the shaded point local coordinate system.

Step 3: Transform the Random Samples to the Shaded Point Local Coordinate System

We already computed a local coordinate Cartesian system in which the normal of the shaded point is aligned with the up vector of that coordinate system. As we know from the lesson on geometry, the three vectors of that Cartesian coordinate system can be used to build a 4x4 matrix (or 3x3 matrix if we only care about directions) that will transform a direction from whatever space that direction is initially in, to the coordinate system from which we built the transformation matrix.



In this particular case, we will use the vectors of Cartesian coordinate system directly without building a matrix (if you are unsure about how this part of the code works, we recommend that you read the lesson on Geometry):

```
001 Vec3f Nt, Nb;
002
003 createCoordinateSystem(hitNormal, Nt, Nb);
004
005 for (uint32_t n = 0; n < N; ++i) {
006     float r1 = distribution(generator); // cos(theta) = N.Light Direction
007     float r2 = distribution(generator);
008     Vec3f sample = uniformSampleHemisphere(r1, r2);
009     Vec3f sampleWorld(
010         sample.x * Nb.x + sample.y * hitNormal.x + sample.z * Nt.x,
011         sample.x * Nb.y + sample.y * hitNormal.y + sample.z * Nt.y,
012         sample.x * Nb.z + sample.y * hitNormal.z + sample.z * Nt.z);
013     ...
014 }
```

Keep in mind that they shading normal N is the up vector. In other words it accounts for the y-axis in the coordinate system. If you look at the way we generated Nt and Nb in the function `createCoordinateSystem`, you will see that when the shading normal is equal to (0,1,0), Nt is aligned with the positive x-axis and Nb with the positive z-axis respectively.

Step 4, 5 and 6: Trace the Indirect Rays

The next step is to trace the ray, and accumulate the returned color to a temporary variable. Once all the rays have been traced then we divide this temporary variable in which the result of all the rays have been accumulated (the amount of light returned along each indirect cast ray), by N, the total number of samples.

```
001 Vec3f indirectDiffuse = 0;
002 for (uint32_t n = 0; n < N; ++i) {
003     float r1 = distribution(generator); // cos(theta) = N.Light Direction
004     float r2 = distribution(generator);
005     Vec3f sample = uniformSampleHemisphere(r1, r2);
006     ...
007     // don't forget to apply cosine law (i.e. multiply by cos(theta) = r1).
008     // we should also divide the result by the PDF (1/2*M_PI) but we can do this after
009     indirectDiffuse += r1 * castRay(hitPoint + sampleWorld * options.bias,
010         sampleWorld, depth + 1, ...);
011 }
012 // divide by N and the constant PDF
013 indirectDiffuse /= N * (1 / (2 * M_PI));
```

Don't forget that the light intensity returned along the direction `sampleWorld` also needs to be multiplied by the cosine of the angle between the shaded normal and the light direction (which is the ray direction in this case). Because our object is diffuse, we need to apply the cosine law (which we introduced in the first lesson on shading). Remember than the random variable `r1` is actually equal to $\cos\theta$ which is the cosine of the angle between N and the ray direction (check the code of the function `uniformSampleHemisphere`). So all there is to do in this case is multiply the result of `castRay` (the amount of light reaching P from the direction `sampleWorld`) by `r1` (line 8).

The generic equation we are trying to solve with this basic form of Monte Carlo integration is:

$$L(P) = \int_{\Omega_N} L(\omega_i)\, BRDF(P, \omega_i, \omega_o)\, \cos(\omega_i, N)\, d\omega = \int_{\Omega_N} L(\omega_i)\, \frac{\rho}{\pi}\, \cos(\omega_i, N)\, d\omega$$

This equation is a simplified form of what we call in CG the **rendering equation**. This is a very important equation which we will introduce more formally in the next lessons. The term $L(P)$ on the left of the equation is the color of the point P that we wish to compute. Technically, the value we are trying to compute is the **radiance** at P. On the right side of this equation, you can find the integral sign. As usual the domain of integration is the hemisphere of directions oriented about the surface normal. In this particular case, we deal with solid angles. Then we have the term $L(\omega_i)$ which is the amount of light illuminating P from the direction ω_i. Then we have the BRDF of the surface P belongs to. The BRDF of a diffuse surface is ρ/π (where ρ is the surface albedo or color if you prefer). Diffuse surface are view independent. In other words the result of this function doesn't depend on the light incoming and view outgoing direction. Though for most BRDFs, the result of the function depends on these two variables, which is why we have to pass them on to the function. Finally, we have the cosine law term, the $\cos(\omega_i, N)$ term. The result of the BRDF should be attenuated by the cosine of the angle between the light incident direction and the surface normal.

Note that in this equation, the BRDF of the shading point and computing the cosine term are things we can easily do by accessing the scene data. Though the term $L(\omega_i)$ is unknown and the way we evaluate it, is by tracing a ray in the scene.

This is a slightly more formal and compact way of looking at the problem we are trying to solve. As mentioned, we will get back to this equation in the next lessons.

> Note that this is also where we should be dividing the result of `castRay` by the random variable PDF (which as we showed earlier, is equal to $1/(2\pi)$). Because the PDF is a constant in this case, we don't need to do this division for each ray (or each sample). We can do it after the loop (line 13). Though we will show later that this can be simplified even further. But don't forget about the division by the PDF!

Step 7: Add the Contribution of the Indirect Diffuse to the Illumination of the Shaded Point

Final touch. Add the result of the indirect diffuse calculation to the total illumination of the shaded point and multiply the sum of the direct and indirect illumination by the object albedo of course.

```
001 | hitColor = (directDiffuse + indirectDiffuse) * object->albedo / M_PI;
```

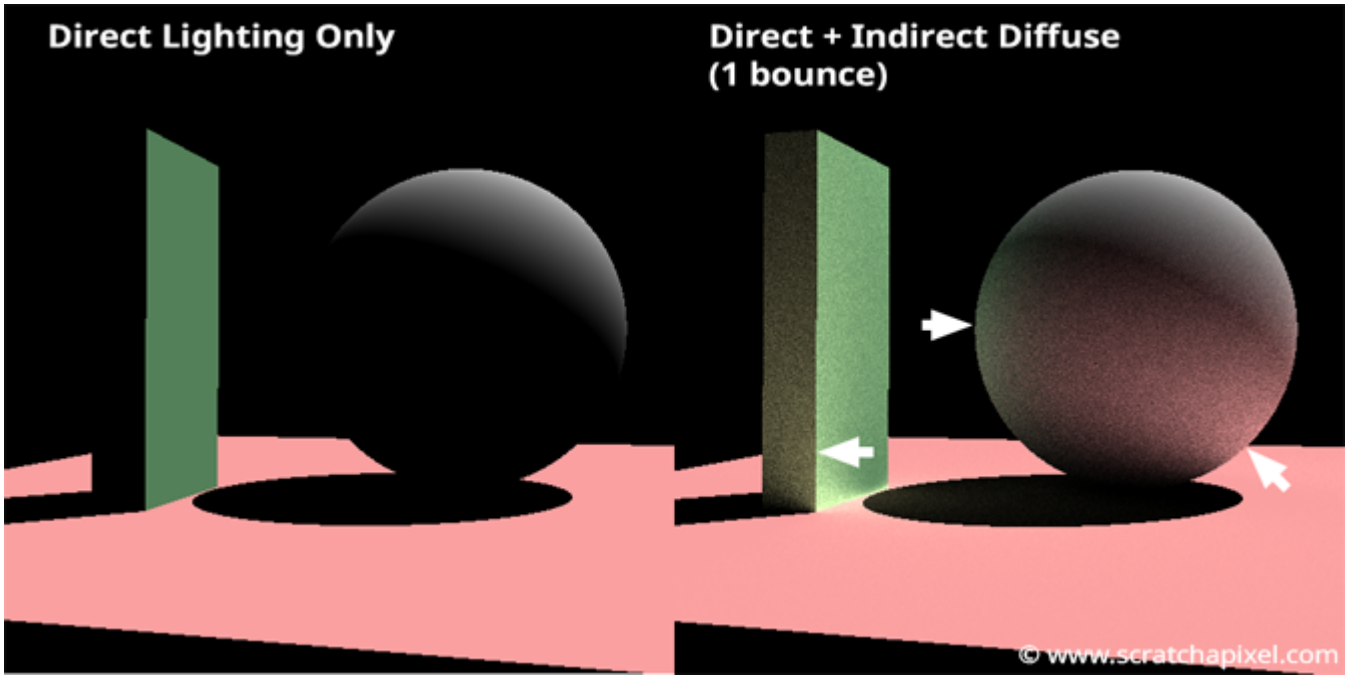
Don't forget that the BRDF of a diffuse surface is ρ/π where ρ stands for the suface albedo. So we need to divide the result by π. Though in the case of the indirect diffuse component, don't forget that we should have divided the result of `castRay` by the PDF of the random variable, which as we showed earlier in this chapter is $1/(2\pi)$. Dividing `indirectDiffuse` by $1/(2\pi)$ is the same as multiplying this value by 2π. And since the albedo is also divided by π we can simplify the code above with:

```
001 | // (indirectDiffuse / (1 / 2 * M_PI) + directDiffuse) * albedo / M_PI
002 | // (2 * M_PI * indirectDiffuse + directDiffuse) * albedo / M_PI
003 | // (2 * indirectDiffuse + directDiffuse / M_PI) * albedo
004 | hitColor = (directDiffuse / M_PI + 2 * indirectDiffuse) * object->albedo;
```

ET VOILA!

Monte Carlo Path Tracing: Results

Let's now see some results. You can find the source code used to render the scene below in the last chapter of this lesson (as usual). In the image below you can see two renders: on the left, global illumination was turned off, and on the right it was turned on. Arrows in the image indicates part of the image in which the effect of other surfaces illuminating other objects in the scene can be clearly scene, such as for example the effect of the floor on the bottom right part of the sphere, or the effect of the green monolith on the left side of the sphere. Notice how the sphere takes on the red or green color of the floor and monolith respectively: the term **color bleeding** if often used to describe this phenomenon (because the color of objects bleed onto others).

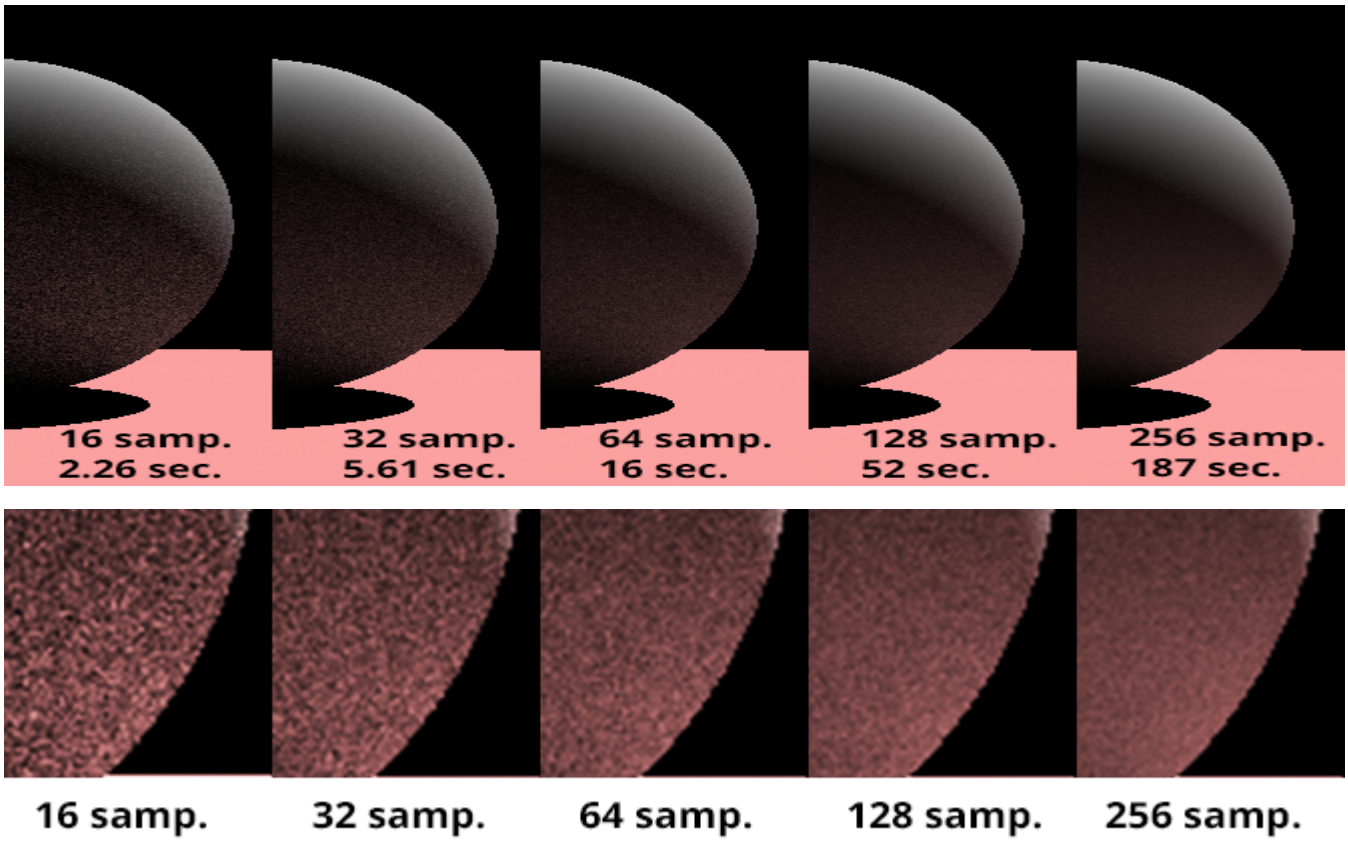


The render time is already significantly slower compared to the time it takes to render direct lighting alone. Thus, in this particular example we have limited the number of bounces to 1, but you can experiment with higher number of bounces (if you are patient enough to wait for the result).

Monte Carlo Path Tracing is Slow and Noisy

Now you as can see in the images below, Monte Carlo path tracing produces noisy images. This is the main problem with this technique. Why? If we go back to our poll example, since each time you want to make estimate the average weight of the adult population of a country you choose randomly say 1,000 persons from that population, each time you will run the poll, you are likely to get 1,0000 totally different people. And thus, each time you compute the average of a new sample, you will get a different number (two samples of 1,000 individuals may as well have the exact same average weight but call this luck). So if you were to compute the average of say 10 groups of 1,000 adults, you you will get 10 different approximations. This is where the noise comes from. There is always more or less variation between what the exact result should be (which is what we want to compute with the integral) and the result of the Monte Carlo integral which is just an approximation. This variation is what produces noise in the image.

How big is the variation depends essentially on the sample size, the number N. By increasing N, we can reduce the noise in the image. The image above shows a close up on the sphere for different values of N. As N increases, you can clearly see that the noise becomes less visible. However, obviously this has a cost, since render time increases with the number of samples used. Monte Carlo theory tells us that to reduce some level of noise (called **variance**) by 2, you need 4 times more samples than the number of samples used in the first place. In other words, to reduce the noise of the first image on the left by 2, you don't need 32 samples but 64 samples, and to halve the noise in the image rendered with 128 samples, you would need 512 samples. You can find why in the lesson Mathematical Foundation of Monte Carlo methods.



Noise is one of the major problems with Monte Carlo path tracing. It is maybe less objectionable on a still image than it is for a sequence of frames. Many of the most recent CG animation feature films such as Finding Dory or Big Hero 6, use a special technique called de-noising. Once rendered, the noise can be partially removed using a special image processing technique.

How Do I Know that My Implementation Does the Right Thing? The Furnace Test

When you implement the technique we described in this lesson, you want to be sure that your implementation is correct. One simple method for testing your implementation is to render a 0.18 grey ball (or 0.5 for that matter it doesn't make much of difference) inside a "white spherical environment". Imagine for instance that your grey sphere is surrounded by a larger white sphere emitting light. In our program, we can simulate this easily by making the `castRay()` to return 1, when rays don't intersect geometry and render the image of grey sphere. What would you expect the result to be? Well, theory tells us that if the object is diffuse then the albedo of an object can be seen as the amount of light it reflects over the amount of light it receives. Clearly, in this setup, each point on the grey sphere receive "white" light from all the directions contained in the hemisphere oriented about the normal at that point. This light or energy is also reflected in all directions in this hemisphere. This is how diffuse surfaces reflect light. Thus, logically if a point receives "white" from all direction and reflects 18% of that incoming light in all directions (if the sphere's albedo is 0.18), then we would expect each point on the sphere to have an intensity of 0.18 exactly.

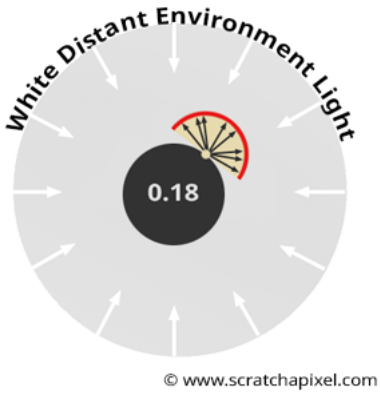


Figure 5: the color of a sphere illuminated by a white spherical light source should equal the sphere's albedo (0.18 in this example).

This is a simple test you can do. Render a 0.18 grey sphere in a white environment. The render of the sphere should look like the render in

figure 6. The sphere should have constant intensity across its surface, and you should read an intensity of 0.18. If it is not the case, then somehow your code is not yet working properly. Of course, if you use Monte Carlo integration to render this image, each pixel in the sphere may have a slightly different intensity, but as N increases, it should converge to 0.18.

> This test in the computer graphics literature, is often referred to as the **furnace** test.

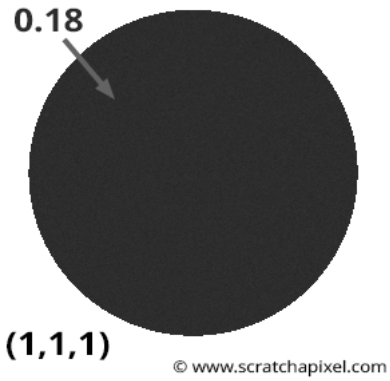


Figure 6: the color of a sphere illuminated by a white spherical light source should equal the sphere's albedo (0.18 in this example).

The name comes from a property of a furnace in thermal equilibrium. When a furnace reaches equilibrium (when the amount of energy received at a point is equal to the amount of energy emitted at that point), its interior has a completely uniform appearance, causing all geometric features to disappear.

Many things can go wrong when you implement global illumination but the most common mistakes are: you forget to divide the illumination function by the PDF of the random variable, the random variables are not generated with the right distribution, you forget to multiply the light intensity (light energy traveling along the light direction which as we mentioned before is called radiance) by the cosine of the angle between the light direction and the surface normal, etc.

```
001 Vec3f castRay(
002     const Vec3f &orig, const Vec3f &dir,
003     ...)
004 {
005     if (trace(orig, dir, objects, isect)) {
006         ...
007         switch (isect.hitObject->type) {
008             case kDiffuse:
009             {
010                 ...
011                 Vec3f indirectLigthing = 0;
012 #ifdef GI
013                 uint32_t N = 128;// / (depth + 1);
014                 Vec3f Nt, Nb;
015                 createCoordinateSystem(hitNormal, Nt, Nb);
016                 float pdf = 1 / (2 * M_PI);
017                 for (uint32_t n = 0; n < N; ++n) {
018                     float r1 = distribution(generator);
019                     float r2 = distribution(generator);
020                     Vec3f sample = uniformSampleHemisphere(r1, r2);
021                     Vec3f sampleWorld(
022                         sample.x * Nb.x + sample.y * hitNormal.x + sample.z * Nt.x,
023                         sample.x * Nb.y + sample.y * hitNormal.y + sample.z * Nt.y,
024                         sample.x * Nb.z + sample.y * hitNormal.z + sample.z * Nt.z);
025                     // don't forget to divide by PDF and multiply by cos(theta)
026                     indirectLigthing += r1 * castRay(hitPoint + sampleWorld *
027                         options.bias,sampleWorld, objects, lights,
028                         options, depth + 1) / pdf;
029                 }
030                 // divide by N
031                 indirectLigthing /= (float)N;
032 #endif
033 
034                 hitColor = (directLighting + indirectLigthing) *
035                     isect.hitObject->albedo / M_PI;
```

```
036                 break;
037             }
038             ...
039         }
040     }
041     else {
042         // return white (as if the grey sphere was
043         // illuminated by a white spherical light)
044         hitColor = 1;
045     }
046 
047     return hitColor;
048 }
049 
050 int main(int argc, char **argv)
051 {
052     ...
053 
054     Sphere *sph = new Sphere(Matrix44f::kIndentiy, 1);
055     sph->albedo = 0.18;
056     objects.push_back(std::unique_ptr<Object>(spy));
057 
058     // finally, render
059     render(options, objects, lights);

    return 0;
}
```

Unbiased Path Tracing: What Does it Mean?

You may have heard the term "biased" or "unbiased" path tracing but what does it mean? These concepts have already been introduced in the lesson Mathematical Foundations of Monte Carlo Methods. You can used different estimators than the one we used in this lesson. The one we used here was:

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

It is a good estimator in the sense that it converges in probability to the exact solution (the actual solution of the integral), but it converges quite slowly. As explained above, each time you want to halve the noise in the image, you need four times more samples. Other estimators exist. They have the advantage of converging faster, but at the price of introducing a "small" bias in the result. In other words, they will provide a result close to the real result, but this result will be slightly above or below the real number. These estimators are said to be "biased". Wen you write a renderer, if being "physically accurate" is not an absolute requirement, it might be better to use a "biased" estimator, as it makes it possible to get an image that is potentially less noisy (or not noisy at all in some cases) at the fraction of the time it would take to render an image of the same quality using an "unbiased" estimator. When you write a production renderer, you often have to find a tradeoff between "accuracy" and speed.

Environment Illumination and Image-Based Illumination

It is actually possible to make a simple change to the code to implement a feature called **environment lighting**. The idea is to simulate somehow the illumination of objects by their environment, which in the real-world can be something like the sky and all the other objects surrounding a given object you are looking at right now. The idea is to surround the CG objects with a spherical light source. This spherical light source can be emitting light uniformly all

across its surface. To evaluate the contribution of this light source, we will use the Monte Carlo integration technique again. For each shaded point (or intersection point hit by a primary ray) we will cast N rays uniformly distributed in the hemisphere oriented about the shading normal. In some cases, these rays will hit an object. In this case, the object this ray intersected is blocking light emitted by the spherical light source to reach the shading point as shown in figure 8. When the ray doesn't intersect any geometry, we can assume that the spherical light and the shading point are visible to each other, and that P is illuminated by that light for the direction defined by the ray (figure 8).

Figure 7: direct illumination of a scene by a white distant environment light.

We assume in this example, that the spherical light source is infinitely far away from the shading points. And thus, the square falloff rule doesn't apply. Such lights are called **distant environment light** and are typically used in CG to simulate the lighting of objects by their distant environment.

We can use this technique to compute how much light from the spherical light actually reaches P. This gives us the image below on the left:

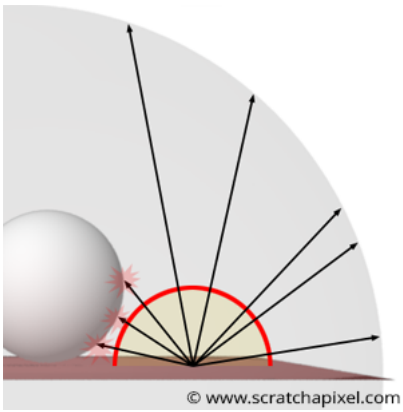
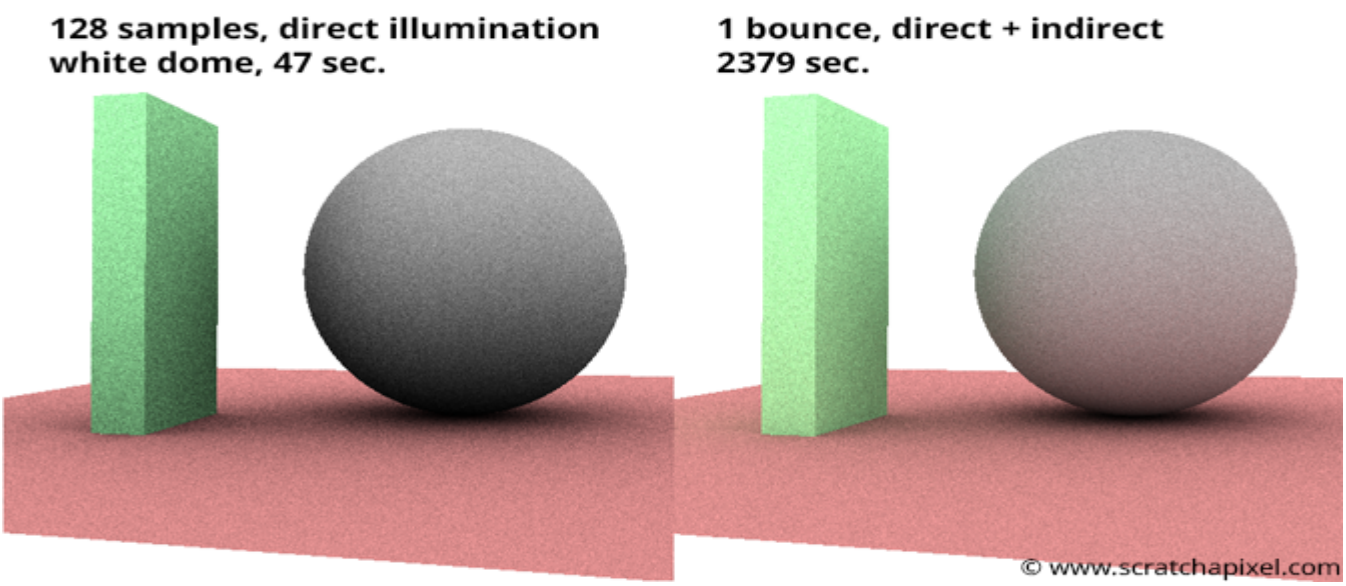


Figure 8: to compute the direct illumination arriving on a point by a distant environment light, we can use Monte Carlo integration again. We cast rays uniformly distributed in the hemisphere of directions. If a ray intersect an object, the shaded point is not illuminated by the sphere in the ray's direction.

Wile we have been using Monte Carlo integration in this particular to compute direct illumination (the illumination of each point in the scene by the spherical light source), we can also activate global illumination and compute the amount of light reflected by objects in the scenes, each time a ray cast from P to compute direct illumination intersects an object. This can become very expensive as the number of bounces increases. Adding direct and indirect illumination using a white environment light can be seen in the image above on the right.



Rather than using a constant color (white in our example) you can also "map" the spherical light with an image. When you trace a ray in a given direction, you then convert the Cartesian coordinates of the ray into spherical coordinates which are then remapped to image coordinates. These coordinates indicate the position of the pixel in the image that illuminates P along the ray direction. We use the image to store real-world lighting information. This technique of illumination is called **image based lighting**. You can find a lesson devoted to this technique in the next section.

So Why the Heck Do We Call that Monte Carlo Path Tracing?

After all these explanations, this should be obvious to you now. Monte Carlo because we use Monte Carlo methods to solve global illumination which itself can be defined mathematically as an integral. This integral means "collect all light illuminating P for all the directions contained in the hemisphere above P". Path tracing, simply because we simulate light paths, the natural path of light rays as they bounce from surface to surface (or interact with various materials: glass, water, specular surfaces, etc.). The technique we described in this lesson is one kind of light transport. It is also called **unidirectional path tracing** because we only follow the path of light rays in one single direction: from the eye back to the light sources. As you may have guessed, there is another algorithm called **bidirectional path-tracing** which consists of tracing some light rays from the eyes to the objects in the scene, other rays from the light sources to the objects in the scene, and connecting the two types of rays to each other to form complete light paths. This method is better to simulate indirect specular reflections or caustics and will be described in the next section.

Congratulation! Knowing so much about rendering already is quite an achievement. In just a few lessons, you have learned about light transport, Monte Carlo integration, BRDF and Path Tracing. These are some of the most dificult concepts to understand in computer graphics.

Conclusion

- The principle of Monte Carlo integration to simulate global illumination is simple.
- Though getting the maths right, can be tricky. It generally requires to know Monte Carlo theory well, what a PDF, a CDF, an estimator and the inversion sampling method are. Though, luckily for you all these things are explained in great details in our two lessons devoted to Monte Carlo methods (theory and practice).
- Ray-Tracing is expensive. Monte Carlo path tracing uses many rays and is thus expensive too, especially as the number of samples used or the number of bounces increases. This show the importance accelerating ray-tracing. We will speak about this in the next section.
- Noise is the main problem of Monte Carlo path-tracing. We will also show in the next section, techniques to reduce noise.
- Monte Carlo integration can also be used to simulate direct lighting from distant environment lights (which is technique image based lighting is based on).

You can find more information on all these topics in the next section.

Reference

The Rendering Equation. Siggraph 1986, James T. Kajiya.