CHAPTER 15

On the Importance of Sampling

Matt Pharr
NVIDIA

ABSTRACT

With the recent arrival of ray tracing to the real-time graphics pipeline, developers are faced with a new challenge: figuring out how to make the most of the rays that they're able to trace. One important question to decide is for which lighting effects to trace rays—choices include shadows, reflections, ambient occlusion, and full global illumination.

Another important question is how to choose *which* rays to trace for the chosen effect; an introduction to that question is the topic of this chapter. In the following, we will see how most lighting calculations in rendering can be interpreted as estimating the values of integrals and how tracing rays is a natural fit to an effective numerical integration technique: Monte Carlo. Given some background in Monte Carlo integration, we then see how well-chosen rays can dramatically improve the speed of convergence, which in turn can either improve overall system performance—by getting the same quality result for fewer rays—or improve image quality—by getting lower error from the same number of rays.

15.1 INTRODUCTION

With the introduction of DirectX Raytracing (DXR) at the 2018 Game Developers Conference and then the launch of NVIDIA's RTX GPUs in the summer of 2018, ray tracing has unequivocally arrived for real-time rendering. This is one of the greatest changes the real-time graphics pipeline has seen: after always offering rasterization as the only visibility algorithm, now a second visibility algorithm has been added—ray tracing.

Ray tracing and rasterization complement each other well. Rasterization remains a high-performance way to perform *coherent* visibility computations: it assumes a single viewpoint (possibly a single homogeneous viewpoint, for an orthographic view), and it regularly samples visibility over a pixel grid. Together, these properties allow high-performance hardware implementations that amortize per-triangle work over multiple pixels and incrementally compute depth and coverage from pixel to pixel.

E. Haines, T. Akenine-Möller (eds.), *Ray Tracing Gems*, https://doi.org/10.1007/978-1-4842-4427-2_15

In contrast, ray tracing allows fully *incoherent* visibility computations. Each ray traced can have an arbitrary origin and direction; the hardware places no restrictions on them.

With ray tracing in hand, the task for developers is to figure out how best to use it. GPU ray tracing hardware provides a few visibility primitives: "What is the first thing visible from this point in this direction?" "Is there anything blocking the straight line segment between these two points?" However, it does not dictate how it should be used for image synthesis—it is up to developers to decide how to do that. In a sense, the situation is similar to programmable shading on GPUs: the hardware provides the basic computational capabilities, and it is up to developers to decide the best way to use those for their applications.

To help motivate some of the trade-offs involved in choosing which rays to trace, we start by looking at a basic ambient occlusion computation through the lens of Monte Carlo integration. We will see how different sampling techniques (and thus different rays traced) lead to different amounts of error in the results before moving on to see the application of some of these ideas to direct illumination from area light sources.

Sampling well for rendering is a complex topic—whole books have been written on the topic and it remains an active area of research. Thus, this chapter can only scratch the surface of the topic, but it includes pointers to resources that provide more information along the way.

15.2 EXAMPLE: AMBIENT OCCLUSION

Most computations related to light and reflection and graphics can be understood as integration problems: for example, we integrate the product of incident light arriving at all directions over the hemisphere at a point with the bidirectional scattering distribution function (BSDF) that describes reflection at the point in order to compute reflected light from a surface.

Monte Carlo integration has been shown to be an effective method for these integration tasks in rendering. It is a statistical technique based on taking a weighted average of random samples of the integrand. Monte Carlo is a good choice for rendering because it works well with high-dimensional integrals (as we end up encountering with global illumination), places few restrictions on the functions to which it can be applied, and only requires point-wise evaluation of them. See the book by Sobol [5] for an approachable introduction to the topic.

Sampling is a perfect fit for ray tracing—it corresponds directly to queries like "is the light source visible in this direction?"or "what is the first visible surface in this direction?"

Here is the definition of the basic Monte Carlo estimator with k samples, which gives a method for computing an approximation to an n-dimensional integral of some function f:

$$E\left[\frac{1}{k}\sum_{i=1}^{k} f(X_i)\right] = \int_{[0,\,1]^n} f(x)\ \mathrm{d}x. \tag{1}$$

On the left-hand side of the equality, we have E, which denotes the *expected value*; the idea is that it indicates that, statistically, the quantity in square brackets is expected to take on the value of the expression on the right-hand side. Sometimes it may be larger and sometimes it may be smaller, but we can imagine that in the limit of more and more values, we expect its average to converge.

The expression inside the square brackets is an average of values of f using a set of independent *random variables* X_i that take on all values in $[0, 1]^n$ with uniform probability. In an implementation, each X_i might just be an n-dimensional random number, but here, writing the Monte Carlo estimator in terms of random variables is what allows rigorous discussion of the expected value.

We have now an easy-to-implement way to estimate the value of any integral. Let us apply it to *ambient occlusion*, a useful shading technique that gives a reasonable approximation to some global lighting effects. We define the ambient occlusion function a at a point P as

$$a(P) = \frac{1}{\pi}\int_{\Omega} v_d(\omega)\cos\theta\ \mathrm{d}\omega, \tag{2}$$

where v_d is a visibility function that is zero if a ray from P in the direction ω is occluded at a distance less than d and one otherwise, where Ω denotes the hemisphere of directions around the surface normal at P, and where the angle θ is measured with respect to the surface normal. The $\frac{1}{\pi}$ term ensures that the value of $a(P)$ is between zero and one.

Consider now the application of the basic Monte Carlo estimator to ambient occlusion. Here, we integrate over the hemisphere rather than a $[0, 1)^n$ domain, but it is not too hard to use a few changes of variables to show that the estimator applies to other integration domains as well. Our estimator is

$$a(P) = E\left[\frac{1}{k}\sum_{i=1}^{k}\frac{1}{\pi} v_d(\omega_i)\cos\theta_i\right], \tag{3}$$

where ω_i are random directions over the hemisphere, each one chosen with uniform probability.

There is a straightforward recipe for choosing directions with this distribution over the hemisphere. Given random numbers ξ_1 and ξ_2 in [0, 1), the following then gives us a direction over the hemisphere centered around the direction (0, 0, 1) (thus, the direction would then need to be transformed to a coordinate frame with the z-axis aligned with the surface normal):

$$(x, y, z) = \left(\sqrt{1-\xi_1^2}\cos(2\pi\xi_2),\ \sqrt{1-\xi_1^2}\sin(2\pi\xi_2),\ \xi_1\right). \tag{4}$$

Figure 15-1 shows a crown model shaded using ambient occlusion, using four samples for the estimator. With just four samples and no denoising, the result is naturally noisy, but we can see that it looks like it is heading in the right direction.

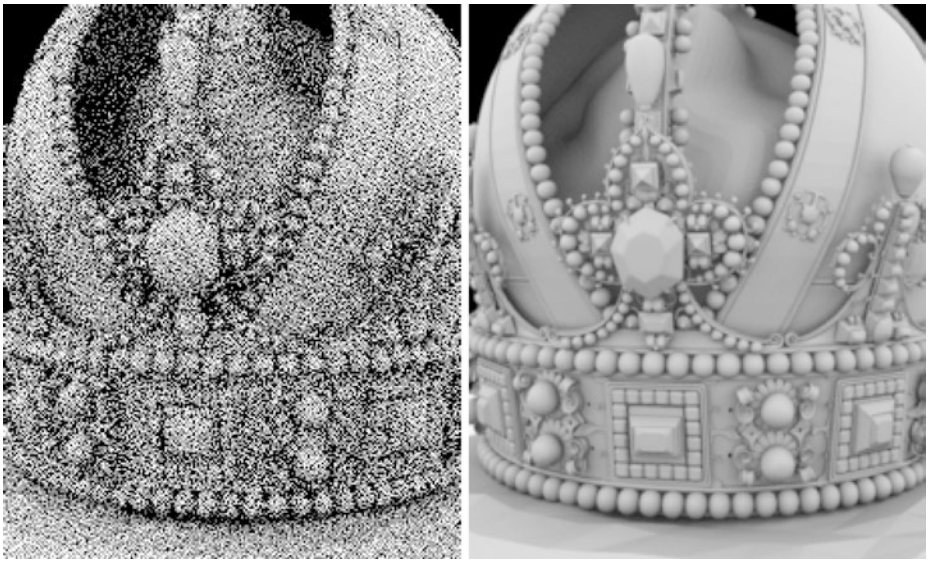

Figure 15-1. *Crown model rendered with ambient occlusion. Left: we traced four random rays per pixel, uniformly distributed over the hemisphere at the visible point. Right: a reference image of the converged solution was rendered with 2048 rays per pixel.*

Another Monte Carlo estimator allows random samples to be taken from nonuniform probability distributions. Here is its definition:

$$E\left[\frac{1}{k}\sum_{i=1}^{k}\frac{f(X_i)}{p(X_i)}\right] = \int_{[0,\,1]^n} f(x)\ \mathrm{d}x. \tag{5}$$

The idea is that now the independent random variables X_i are distributed according to some possibly nonuniform distribution $p(x)$. Due to the division by $p(X_i)$, everything works out: when we are more likely to take samples in some part of the domain, $p(X_i)$ is relatively large and the contribution of those samples is reduced. Conversely, choosing a sample with a low probability will happen less frequently than it would with uniform sampling, but those samples contribute more since their $p(X_i)$ value is relatively small. Note that in our example no samples will have $p(X_i) = 0$.

Why might we want to sample nonuniformly like this? We can see why by considering ambient occlusion again. There is a sampling recipe that takes cosine-distributed samples on the hemisphere (again, centered around (0, 0, 1)):

$$(x, y, z) = \left(\sqrt{\xi_1}\cos(2\pi\xi_2), \sqrt{\xi_1}\sin(2\pi\xi_2), \sqrt{1-\xi_1}\right). \tag{6}$$

Again, it takes two independent uniform random samples ξ_1 and ξ_2, each in [0, 1), and transforms them. It turns out that $p(\omega) = \cos\theta/\pi$, where the π term is necessary for normalization.

Pulling it all together, we have the following estimator for the ambient occlusion integral if we use cosine-distributed samples ω_i:

$$a(P) = E\left[\frac{1}{k}\sum_{i=1}^{k}\frac{1}{\pi}\frac{v(\omega_i)\cos\theta_i}{\cos\theta_i / \pi}\right] = E\left[\frac{1}{k}\sum_{i=1}^{k} v(\omega_i)\right]. \tag{7}$$

Because we could generate rays with probability exactly proportional to $\cos\theta$, the cosine terms cancel out. In turn, every ray that we sample has the same contribution to the estimate—either zero or one.[1]

The implementation is straightforward:

```
1 float ao(float3 p, float3 n, int nSamples) {
2    float a = 0;
3    for (int i = 0; i < nSamples; ++i) {
4      float xi[2] = { rng(), rng() };
5      float3 dir(sqrt(xi[0]) * cos(2 * Pi * xi[1]),
6                 sqrt(xi[0]) * sin(2 * Pi * xi[1]),
7                 sqrt(1 - xi[0]));
8      dir = transformToFrame(n, dir);
9      if (visible(p, dir)) a += 1;
10   }
11   return a / nSamples;
12 }
```

[1] If you have implemented screen-space ambient occlusion, it is likely that you are already using this approach, though perhaps now it is easier to understand why it is worth doing so.

Figure 15-2 shows the crown model again, now comparing uniform sampling to cosine-distributed sampling. Cosine-distributed sampling has visibly lower error. Why might this be?

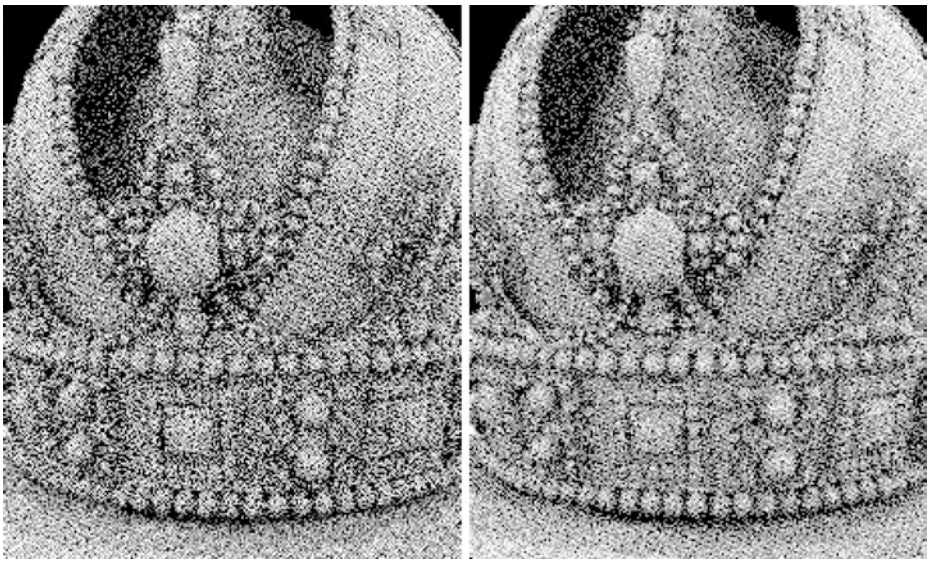

Figure 15-2. *Crown model rendered with ambient occlusion: uniform sampling (left) and cosine-weighted sampling (right). Both used four rays per pixel. Cosine-weighted sampling has nearly 30% lower average pixel error than uniform sampling, which is reflected in its image having noticeably less noise.*

With uniformly distributed sampling, some of the rays turn out to have an insignificant contribution. Consider a ray close to the horizon: its value of $\cos\theta$ will be close to zero, and effectively, we learn little by tracing the ray. Its contribution to the sum in the estimator will either be zero or minimal. Put another way, we do just as much work to trace those rays as all the other rays, but we do not get much out of them. The difference in the amount of computation required to sample rays between the two techniques is negligible, so there is no reason not to use the more effective sampling technique.

This general technique, sampling from a distribution that is similar to the integrand, is called *importance sampling* and is an important technique for efficient Monte Carlo integration in rendering. The closer a match $p(x)$ is to $f(x)$, the better the results. However, if $p(x)$ does not match $f(x)$ well, error will increase as the encountered ratios $f(x)/p(x)$ oscillate between minuscule and huge values. As long as $p(x) > 0$ whenever $f(x) \neq 0$, the result will still be correct in the limit, though the error may be high enough for that to be a small consolation.

15.3 UNDERSTANDING VARIANCE

A concept called *variance* is useful for characterizing the expected error in Monte Carlo integration. The variance of a random variable X is defined in terms of another expectation:

$$V[X] \equiv E\left[\left(X - E[X]\right)^2\right] = E[X^2] - E[X]^2. \tag{8}$$

Variance is thus a measure of the squared difference between a random variable and its expected value (i.e., its average). In other words, if a random variable has low variance, then most of the time its value is close to its average (and the converse if variance is high).

If we can accurately compute the expectation of a random variable (e.g., using Monte Carlo integration with a large number of samples), we can compute an estimate of the variance directly using Equation 8.

We can also estimate the variance: given a number of independent values of a random variable, we can compute the *sample variance* from them using Equation 8 with a small adjustment. The following code illustrates the computation:

```
1 float estimate_sample_variance(float samples[], int n) {
2    float sum = 0, sum_sq = 0;
3    for (int i = 0; i < n; ++i) {
4      sum += samples[i];
5      sum_sq += samples[i] * samples[i];
6    }
7    return sum_sq / (n * (n - 1))) -
8           sum * sum / ((n - 1) * n * n);
9 }
```

Note that it is not necessary to store all the samples: sample variance can also be computed incrementally by keeping track of the sum and squared sum of the values of the random variable and the total number of samples.

One challenge with the sample variance is that it has variance itself: if we happened to have a number of similar sample values even though the underlying estimator had high variance, we would compute a much-too-low estimate of the sample variance.

Variance is a particularly useful concept in Monte Carlo integration, as there is a fundamental relationship between variance and the number of samples taken: *for random samples, variance decreases linearly with the number of samples taken.*[2]

[2]Variance can decrease even faster with certain carefully constructed sampling patterns, especially if the integrand is smooth, though we ignore that for the discussion here.

Thus, the good news is that if we would like to cut the variance in half, we can expect that taking twice as many samples (i.e., tracing twice as many rays) will do just that. Unfortunately, since variance is effectively squared error, that means that cutting error in half requires *four* times as many samples.

This relationship between variance and the number of samples taken helps explain a few things about interactive ray tracing. On one hand, it helps us understand why images improve so much going from one sample per pixel to two, and then to three and more. It is easy to double the number of samples when you have only taken one, and we know that doing so will cut variance in half.

On the other hand, this property also explains why more rays are not always the solution: if we have traced 128 rays in a pixel and still have 2× more variance than we would like, we need 128 more of them to take care of that. It gets even worse if one has an image with thousands of samples per pixel that is still noisy! It is easy to see the value of denoising algorithms in this light; they are a much more effective way to take care of lingering noise than more rays once a reasonable number of rays have been traced.

We computed the average sample variance of all the pixels in the crown renderings. The image of ambient occlusion with uniformly sampled directions (Figure 15-2, left) has an average variance of 0.0972, and the image with cosine-weighted directions (Figure 15-2, right) has average variance 0.0508. The ratio between these variances is approximately 1.91. Thus, we can expect that if we trace 1.91× more rays with uniform sampling than with cosine-weighted sampling, we will get results of roughly equal quality.

We traced four rays per intersection before. Figure 15-3 shows that having $1.91 \times 4 \approx 8$ uniformly distributed directions at each intersection gives similar results to using four cosine-weighted directions. The images appear to have similar quality, and the image with eight uniformly sampled directions per pixel has average pixel variance of 0.0484, which is just slightly better than with four cosine-weighted rays.

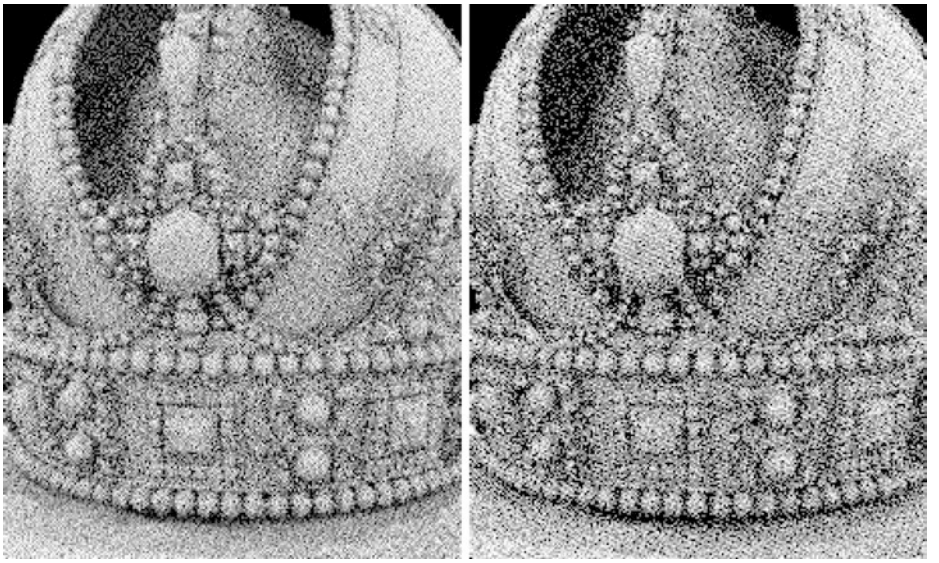

Figure 15-3. *Because variance decreases linearly with sample count, we can accurately estimate how many more samples will be necessary to reduce measured variance a certain amount. We compare the crown with eight uniformly distributed samples (left) and four cosine-distributed samples (right). The variance in both images is nearly the same, even though the one on the right required tracing half as many rays.*

Estimates of variance can also be used to adjust filter kernel widths when denoising: where the variance is low, then not much filtering is needed, but where it is high, a wide filter is likely a good idea. The earlier caveats about the variance in estimates of sample variance apply here: in practice, it is usually a good idea to filter the variance estimates across a group of nearby pixels or temporally over multiple frames in order to reduce the error in the variance estimate.

Estimates of variance can also be a good guide for adaptive sampling algorithms, in which we are trying to decide where more rays should be traced. Indeed, if we can choose the pixels with the highest ratio of variance to number of samples already taken, then we know that we are getting the most out of our additional rays: given the linear decrease in variance with more rays, those rays will have the greatest impact on variance reduction across the whole image.[3]

[3]It turns out that driving adaptive sampling based on sampled values like this causes the Monte Carlo estimator to become *biased* [3], which means that it does not converge in quite the way that we have described so far. The root issue is essentially that error in the estimated sample variance is not the true error.

15.4 DIRECT ILLUMINATION

Another important integral in rendering comes from the *surface scattering equation*, which gives the scattered radiance at a point P in a direction ω_o due to the incident radiance function $L_i(P,\omega)$ and the BSDF $f(\omega \rightarrow \omega_o)$:

$$L_o(P, \omega_o) = \int_{\Omega} L_i(P, \omega) f(\omega \rightarrow \omega_o) \cos\theta \, \mathrm{d}\omega. \tag{9}$$

In this section, we consider the effect of a few different sampling choices when estimating the value of this integral and measure their effect on variance.

Ideally, we would like to be able to sample directions ω proportionally to the value of the product of L_i, f, and $\cos\theta$. In general, this is difficult to do, especially because the incident radiance function generally is not available in closed form—we need to trace rays to evaluate it.

Here, we make a few simplifications. First, we only consider the incident light from emitters in the scene and ignore indirect illumination. Second, we only look at the effect of various choices in sampling proportional to L_i. Note that the second simplification absolutely should not be used in practice: it is imperative to also sample from the BSDF and to use a powerful variance reduction technique called *multiple importance sampling* to weight the samples [6].

With those simplifications, we are left with the task of computing the value of the following Monte Carlo estimator:

$$L_o(P, \omega_o) = E\left[\frac{1}{k}\sum_{i=1}^{k} \frac{L_i(P, \omega_i) f(\omega_i \rightarrow \omega_o) \cos\theta_i}{p(\omega_i)}\right], \tag{10}$$

where the ω_i have been sampled from some distribution $p(\omega)$. Note that if we only consider direct illumination, there is no reason to sample a direction that definitely does not intersect a light source. Thus, a reasonable strategy is to sample according to a distribution over the surface of the light, to choose a point on the light source, and then to set the direction ω_i as the direction from P to the sampled point.

For a spherical emitter, a straightforward approach is to sample points over the entire surface of the sphere. The following recipe takes a pair of uniform samples ξ_1 and ξ_2 and uniformly samples points on the unit sphere at the origin:

$$\begin{aligned} z &= 1-2\xi_1, \\ x &= \sqrt{1-z^2}\cos\left(2\pi\xi_2\right), \\ y &= \sqrt{1-z^2}\sin\left(2\pi\xi_2\right). \end{aligned} \tag{11}$$

Figure 15-4 shows how this approach works in a two-dimensional setting. A problem is evident: more than half of the circle is not visible to a point outside of it, and thus all the samples taken on the backside of the circle with respect to the point lead to wasted rays, because other parts of the circle will occlude the sampled points from the point *P*. The analogous case is true in three dimensions.

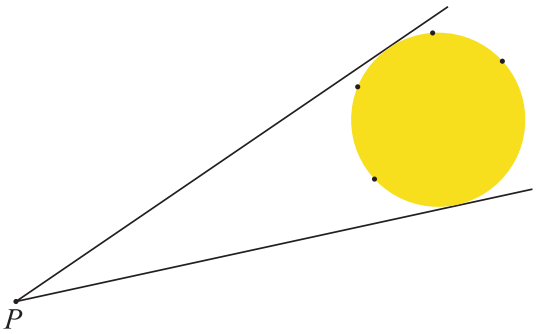

Figure 15-4. *When sampling points on a spherical light source (yellow circle), at least half of the sphere as seen from a point P outside the sphere is occluded. Sampling points uniformly over the surface of the sphere, as shown here, is inefficient because all the samples on the back side of the sphere are occluded by other parts of the sphere and thus are not useful.*

A better sampling strategy is to bound the sphere with a cone from the point *P* and uniformly sample within the cone to choose points on the sphere. Doing so ensures that all the samples are potentially visible to the point (though they still may be occluded by other objects in the scene.) The recipe for sampling uniformly in a cone with angle θ is given in Chapter 16, "Sampling Transformations Zoo," but we repeat it here:

$$\begin{aligned} \cos\theta' &= \left(1-\xi_1\right)+\xi_1\cos\theta, \\ \phi &= 2\pi\xi_2, \end{aligned} \tag{12}$$

where θ' is an angle measured with respect to the cone axis with range $[0, \theta)$ and ϕ is an angle between 0 and 2π that defines a rotation around the cone axis. Figure 15-5 illustrates this technique.

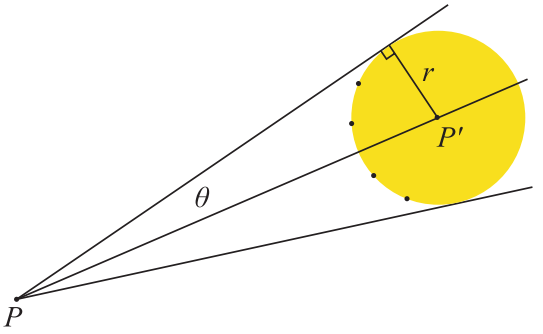

Figure 15-5. *If we compute the angle θ of a cone that bounds a spherical emitter as seen from a point P, then if we sample directions within the cone with uniform probability, we can sample points on the emitter (black dots) that are not on the back side of it with respect to P.*

The improved sampling strategy makes a big difference; images are shown in Figure 15-6. With four rays per pixel, the average pixel variance when sampling the spherical emitters uniformly is 0.0787. Variance is 0.0248, or 3.1× lower, when sampling the cone. As we saw with ambient occlusion, equivalently we can say that 3.1× more rays would need to be traced to generate a result with the same quality if uniform sampling was used rather than sampling within the cone.

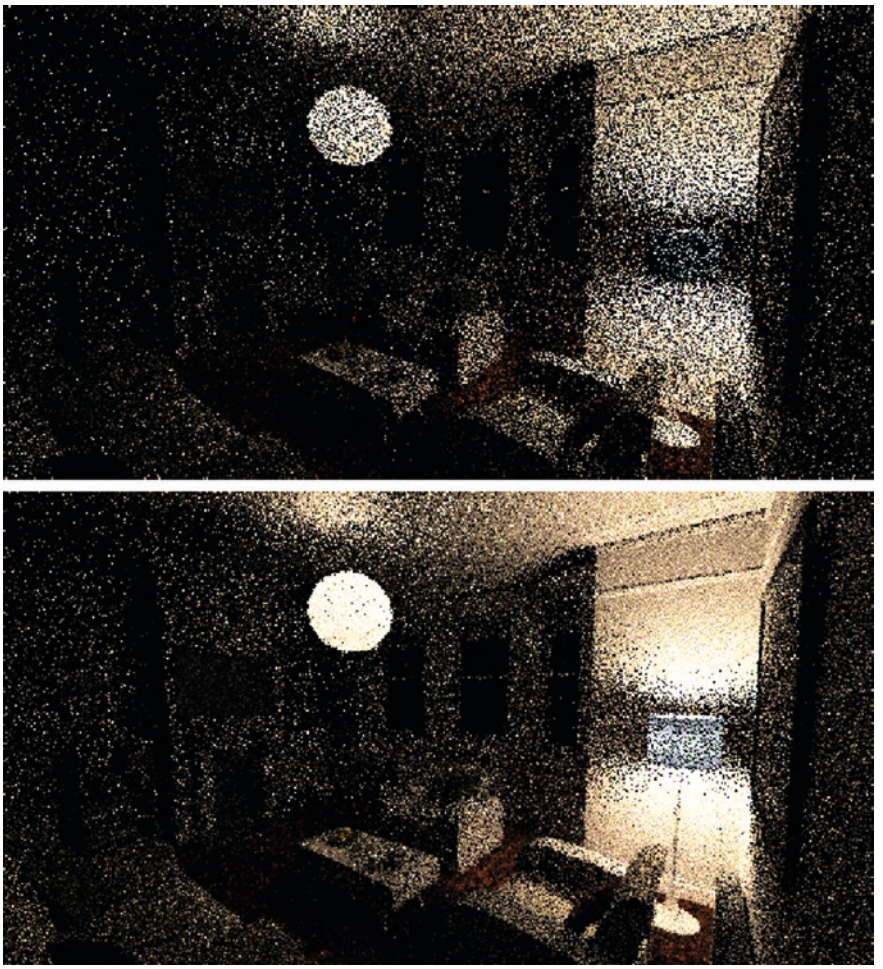

Figure 15-6. *White Room scene at nighttime, with two spherical light sources, rendered with four samples per pixel. Top: uniform sampling of the spherical light sources. Bottom: sampling within the cone subtended from each point being illuminated. Variance is 3.1× lower in the bottom image for the same number of rays traced, thanks to a better sampling method being used. (Scene courtesy of Jay Hardy, under a CC-BY license.)*

As a last example, we show that choosing *which* light to sample makes a big difference with variance as well.

Given a scene with two light sources, such as White Room, the natural thing to do is to trace half of the rays to one light and half to the other. However, consider a point close to one of the two light sources (e.g., on the wall above the floor lamp on the right). It is visually evident that the light source on the ceiling does not contribute as much light to the wall as the light source right next to it. In turn, that means that rays traced to the ceiling light will have a much lower contribution than rays traced to the closer light—exactly the same situation as with ambient occlusion and rays close to the horizon.

If we instead choose which light to sample according to a probability that accounts for its distance to the receiving point and the emitted power, variance is further reduced.[4] Figure 15-7 shows the results. Adapting the probability of sampling lights to their estimated contribution makes another significant improvement: average pixel variance is 0.00921, which is a 2.7× reduction from sampling lights with uniform probability (which had average pixel variance of 0.0248). Together, these two sampling improvements reduced variance by an overall factor of 8.5×.

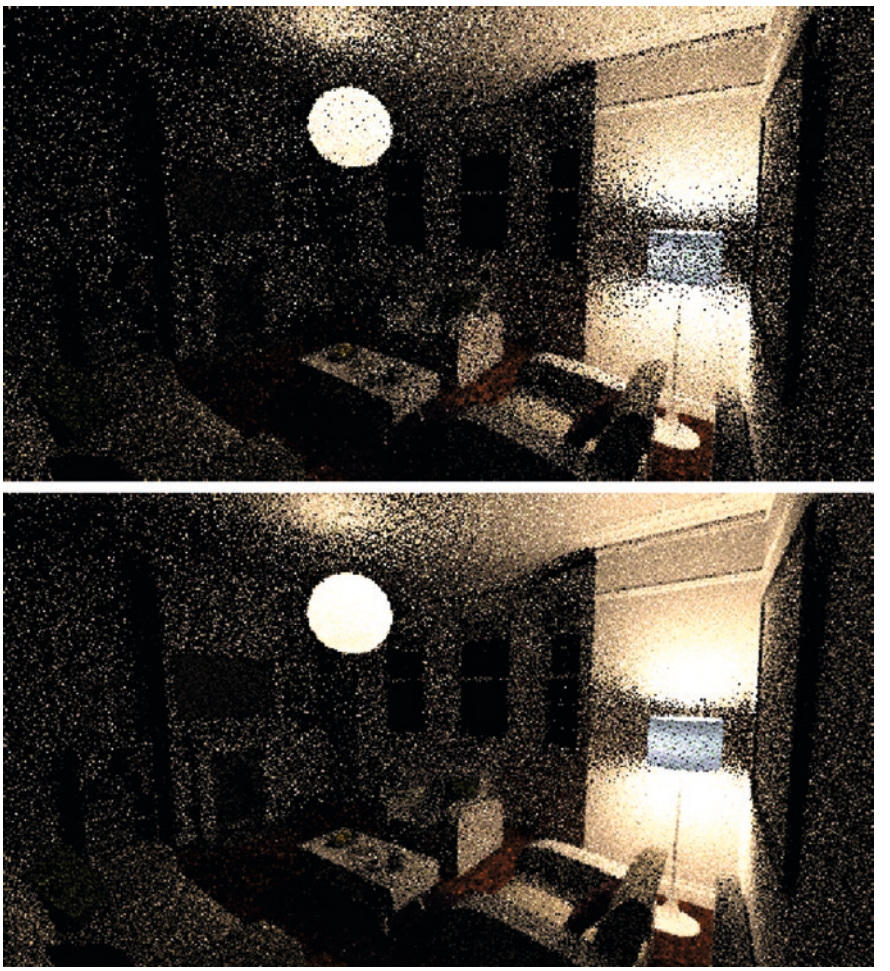

Figure 15-7. *White Room scene at nighttime, comparing different approaches of choosing which light to sample for illumination. Top: lights are sampled with uniform probability. Bottom: lights are sampled with probability proportional to an estimate of the illumination that they cast at the point where reflection is being computed. Variance is reduced by 2.7× by the latter technique. (Scene courtesy of Jay Hardy, under a CC-BY license.)*

[4]See Conty Estevez and Kulla's paper [1], which describes the algorithm we implemented here, as well as Chapter 18, "Importance Sampling of Many Lights on the GPU," where this topic is explored in detail.

15.5 CONCLUSION

We hope that this chapter has left the reader with a basic understanding of the importance of the details of sampling and, more importantly, an understanding of why it is worth sampling well. It is easy to sample inefficiently, but it is not that much harder to sample well. We showed instances of reductions in variance by factors ranging from nearly 2× to 8.5×, purely thanks to more careful sampling and tracing more useful rays.

Given the connection between variance and sample count, another way to look at these results is that if you *do not* sample well, it is more or less the same as having a GPU that is running at $\frac{1}{2}$ to $\frac{1}{8}$ of the actual performance it offers!

This chapter only scratched the surface of how to sample well in ray tracing; for example, we did not discussed how to sample according to the distributions defined by BSDFs or how to apply multiple importance sampling, an important variance reduction technique. See Chapter 28, "Ray Tracing Inhomogeneous Volumes"; Chapter 18, "Importance Sampling of Many Lights on the GPU"; and Chapter 16, "Sampling Transformations Zoo," in this volume for more information on these topics. Furthermore, we did not discuss the substantial error reduction that can be achieved from using more uniformly distributed samples; see Keller's survey [2] for more information about one such approach. Another useful resource for all these topics is the book *Physically Based Rendering* [4], which is now freely available in an online edition.

REFERENCES

[1] Conty Estevez, A., and Kulla, C. Importance Sampling of Many Lights with Adaptive Tree Splitting. *Proceedings of the ACM on Computer Graphics and Interactive Techniques 1*, 2 (2018), 25:1–25:17.

[2] Keller, A. Quasi-Monte Carlo Image Synthesis in a Nutshell. In *Monte Carlo and Quasi-Monte Carlo Methods 2012*. Springer, 2013, pp. 213–249.

[3] Kirk, D., and Arvo, J. Unbiased Sampling Techniques for Image Synthesis. *Computer Graphics (SIGGRAPH) 25*, 4 (1991), 153–156.

[4] Pharr, M., Jakob, W., and Humphreys, G. *Physically Based Rendering: From Theory to Implementation*, third ed. Morgan Kaufmann, 2016.

[5] Sobol, I. M. *A Primer for the Monte Carlo Method*. CRC Press, 1994.

[6] Veach, E., and Guibas, L. J. Optimally Combining Sampling Techniques for Monte Carlo Rendering. In *Proceedings of SIGGRAPH* (1995), pp. 419–428.