CHAPTER FOURTEEN

14 MONTE CARLO INTEGRATION II: IMPROVING EFFICIENCY

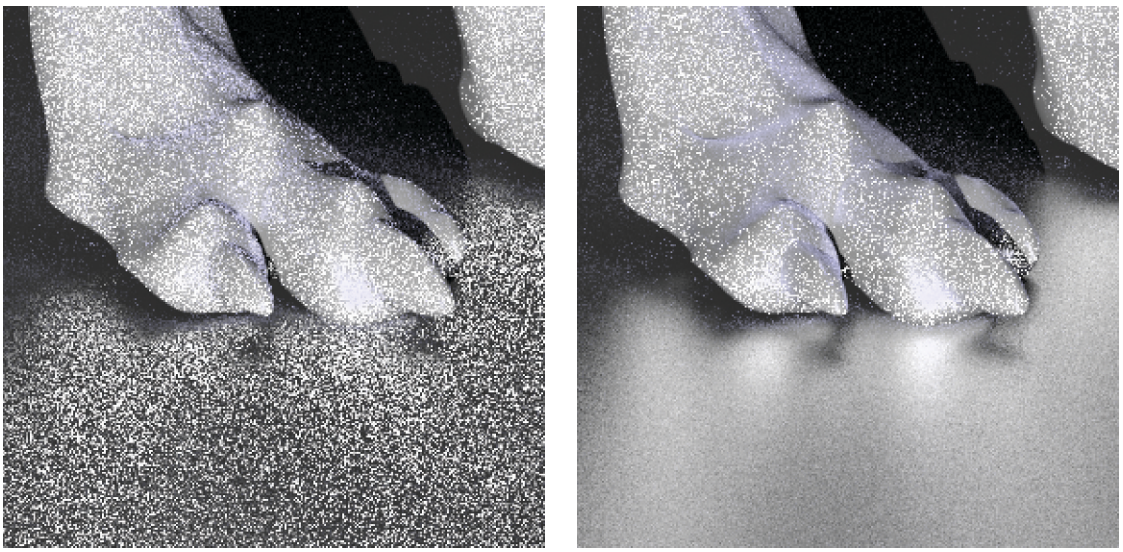
Variance in Monte Carlo ray tracing manifests itself as noise in the image—an unpleasant artifact (Figure 14.1). The battle against variance is the basis of most of the work in optimizing Monte Carlo. Recall that Monte Carlo's convergence rate means that it is necessary to quadruple the number of samples in order to reduce the variance by half. Because the run time of the estimation procedure is proportional to the number of samples, the cost of reducing variance can be high. This chapter will develop the theory and practice of techniques for improving the efficiency of Monte Carlo integration without necessarily increasing the number of samples.

The *efficiency* of an estimator F is defined as

$$\epsilon[F] = \frac{1}{V[F]T[F]},$$

where $V[F]$ is its variance and $T[F]$ is the running time to compute its value. According to this metric, an estimator F_1 is more efficient than F_2 if it takes less time to produce the same variance, or if it produces less variance in the same amount of time.

One of the techniques that has been most effective for improving efficiency for rendering problems is a method called *importance sampling*. Recall that there is some freedom in the choice of distribution $p(x)$ to use as the sample distribution for the Monte Carlo estimator in Equation (13.3). It turns out that choosing a sampling distribution that is similar in shape to the integrand leads to reduced variance. This technique is called importance sampling because samples tend to be taken in "important" parts of the function's domain, where the function's value is relatively large.



(a) (b)

Figure 14.1: (a) A scene with glossy reflection, where variance from estimating the value of the scattering equation at each pixel is visually disturbing. (b) An image computed with the variance reduction techniques described in this chapter; it has substantially less variance but uses the same amount of computation.

The first half of this chapter discusses importance sampling and a number of other techniques for improving the efficiency of Monte Carlo. The second half of the chapter then derives techniques for generating samples according to the distributions of BSDFs, light sources, and functions related to volume scattering so that they can be used as sampling distributions for importance sampling. These sampling methods will be used throughout Chapters 15 through 17.

14.1 RUSSIAN ROULETTE AND SPLITTING

Russian roulette and splitting are two related techniques that can improve the efficiency of Monte Carlo estimates by increasing the likelihood that each sample will have a significant contribution to the result. Russian roulette addresses the problem of samples that are expensive to evaluate but make a small contribution to the final result. We would like to avoid the work of evaluating these samples while still computing a correct estimator.

As an example, consider the problem of estimating the direct lighting integral, which gives reflected radiance at a point due to direct lighting from the light sources in the scene, L_{d}:

$$L_{\mathrm{o}}(\mathrm{p}, \omega_{\mathrm{o}}) = \int_{S^2} f_{\mathrm{r}}(\mathrm{p}, \omega_{\mathrm{o}}, \omega_{\mathrm{i}})\, L_{\mathrm{d}}(\mathrm{p}, \omega_{\mathrm{i}})\, |\cos\theta_i|\, \mathrm{d}\omega_{\mathrm{i}}.$$

Assume that we have decided to take $N = 2$ samples from some distribution $p(\omega)$ to compute the estimator

$$\frac{1}{2}\sum_{i=1}^{2}\frac{f_{\mathrm{r}}(\mathrm{p},\omega_{\mathrm{o}},\omega_i)\,L_{\mathrm{d}}(\mathrm{p},\omega_i)\,|\cos\theta_i|}{p(\omega_i)}.$$

Most of the computation expense of each of the terms of the sum comes from tracing a shadow ray from the point p to see if the light source is occluded as seen from p.

For all of the directions ω_i where the integrand's value is clearly zero (e.g., because $f_{\mathrm{r}}(\mathrm{p},\omega_{\mathrm{o}},\omega_i)$ is zero for that direction), we can skip the work of tracing the shadow ray, since tracing it won't change the final value computed. Russian roulette makes it possible to also skip tracing rays when the integrand's value is very low but not necessarily zero, while still computing the correct value on average. For example, we might want to avoid tracing rays where $f_{\mathrm{r}}(\mathrm{p},\omega_{\mathrm{o}},\omega_i)$ is small, or when ω_i is close to the horizon and thus $|\cos\theta_i|$ is small. Of course, these samples just can't be ignored completely, since the estimator would then consistently underestimate the correct result.

To apply Russian roulette, we select some termination probability q. This value can be chosen in almost any manner; for example, it could be based on an estimate of the value of the integrand for the particular sample chosen, increasing as the integrand's value becomes smaller. With probability q, the integrand is not evaluated for the particular sample and some constant value c is used in its place ($c = 0$ is often used). With probability $1 - q$, the integrand is still evaluated but is weighted by a term, $1/(1 - q)$, that effectively accounts for all of the samples that were skipped:

$$F' = \begin{cases} \frac{F - qc}{1-q} & \xi > q \\ c & \text{otherwise.} \end{cases}$$

The expected value of the resulting estimator is the same as the expected value of the original estimator:

$$E[F'] = (1-q)\left(\frac{E[F]-qc}{1-q}\right) + qc = E[F].$$

Russian roulette never reduces variance. In fact, unless somehow $c = F$, it will always increase variance. However, it can improve efficiency if probabilities are chosen so that samples that are likely to make a small contribution to the final result are skipped.

One pitfall is that poorly chosen Russian roulette weights can substantially increase variance. Imagine applying Russian roulette to all of the camera rays with a termination probability of .99: we'd only trace 1% of the camera rays, weighting each of them by $1/.01 = 100$. The resulting image would still be "correct" in a strictly mathematical sense, although visually the result would be terrible: mostly black pixels with a few very bright ones. One of the exercises at the end of the next chapter discusses this problem further and describes a technique called *efficiency-optimized Russian roulette* that tries to set Russian roulette weights in a way that minimizes the increase in variance.

14.1.1 SPLITTING

While Russian roulette reduces the effort spent evaluating unimportant samples, splitting increases the number of samples taken in order to improve efficiency. Consider again the problem of computing reflection due only to direct illumination. Ignoring pixel filtering, this problem can be written as a double integral over the area of the pixel A and over

the sphere of directions $\mathcal{S}^2$ at the visible points on surfaces at each (x, y) pixel position, where $L_{\mathrm{d}}(x, y, \omega)$ denotes the exitant radiance at the object visible at the position (x, y) on the image due to incident radiance from the direction ω:

$$\int_A \int_{\mathcal{S}^2} L_{\mathrm{d}}(x, y, \omega)\, \mathrm{d}x\, \mathrm{d}y\, \mathrm{d}\omega.$$

The natural way to estimate the integral is to generate N samples and apply the Monte Carlo estimator, where each sample consists of an (x, y) image position and a direction ω toward a light source. If there are many light sources in the scene, or if there is an area light casting soft shadows, tens or hundreds of samples may be needed to compute a result with an acceptable variance level. Unfortunately, each sample requires that two rays be traced through the scene: one to compute the first visible surface from position (x, y) on the image plane, and one a shadow ray along ω to a light source.

The problem with this approach is that if $N = 100$ samples are taken to estimate this integral, then 200 rays will be traced: 100 camera rays and 100 shadow rays. Yet, 100 camera rays may be many more than are needed for good pixel antialiasing and thus may make relatively little contribution to variance reduction in the final result. Splitting addresses this problem by formalizing the approach of taking multiple samples in some of the dimensions of integration for each sample taken in other dimensions.

With splitting, the estimator for this integral can be written taking N image samples and M light samples per image sample:

$$\frac{1}{N}\frac{1}{M}\sum_{i=1}^{N}\sum_{j=1}^{M}\frac{L(x_i, y_i, \omega_{i,j})}{p(x_i, y_i)\, p(\omega_{i,j})}.$$

Thus, we could take just 5 image samples, but take 20 light samples per image sample, for a total of 105 rays traced, rather than 200, while still taking 100 area light samples in order to compute a high-quality soft shadow.

14.2 CAREFUL SAMPLE PLACEMENT

A classic and effective family of techniques for variance reduction is based on the careful placement of samples in order to better capture "important" features of the integrand (or, more accurately, to be less likely to miss important features). These techniques are complementary to techniques like importance sampling and are used extensively in `pbrt`. Indeed, one of the tasks of `Samplers` in Chapter 7 was to generate well-distributed samples for use by the integrators for just this reason, although at the time we offered only an intuitive sense of why this was worthwhile. Here we will formally justify that extra work.

14.2.1 STRATIFIED SAMPLING

Stratified sampling was first introduced in Section 7.3, and we now have the tools to motivate its use. Stratified sampling works by subdividing the integration domain Λ into n nonoverlapping regions $\Lambda_1, \Lambda_2, \ldots, \Lambda_n$. Each region is called a *stratum*, and they

Sampler 340

must completely cover the original domain:

$$\bigcup_{i=1}^{n} \Lambda_i = \Lambda.$$

To draw samples from Λ, we will draw n_i samples from each Λ_i, according to densities p_i inside each stratum. A simple example is supersampling a pixel. With stratified sampling, the area around a pixel is divided into a $k \times k$ grid, and a sample is drawn from each grid cell. This is better than taking k^2 random samples, since the sample locations are less likely to clump together. Here we will show why this technique reduces variance.

Within a single stratum Λ_i, the Monte Carlo estimate is

$$F_i = \frac{1}{n_i} \sum_{j=1}^{n_i} \frac{f(X_{i,j})}{P_i(X_{i,j})},$$

where $X_{i,j}$ is the jth sample drawn from density p_i. The overall estimate is $F = \sum_{i=1}^{n} v_i F_i$, where v_i is the fractional volume of stratum i ($v_i \in (0, 1]$).

The true value of the integrand in stratum i is

$$\mu_i = E\left[f\left(X_{i,j}\right)\right] = \frac{1}{v_i} \int_{\Lambda_i} f(x)\, \mathrm{d}x,$$

and the variance in this stratum is

$$\sigma_i^2 = \frac{1}{v_1} \int_{\Lambda_i} \left(f(x) - \mu_i\right)^2 \mathrm{d}x.$$

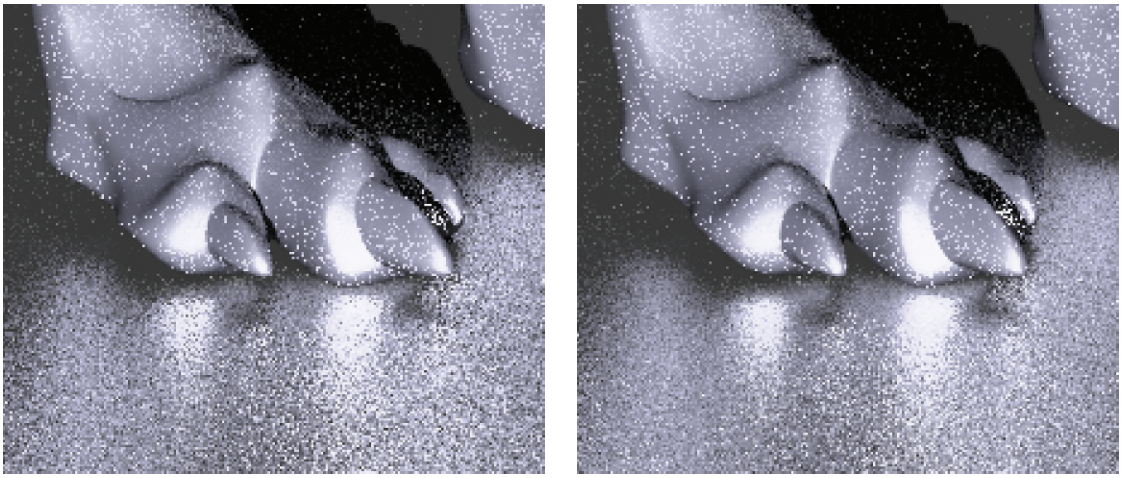
Thus, with n_i samples in the stratum, the variance of the per-stratum estimator is σ_i^2/n_i. This shows that the variance of the overall estimator is

$$\begin{aligned} V[F] &= V\left[\sum v_i F_i\right] \\ &= \sum V\left[v_i F_i\right] \\ &= \sum v_i^2 V\left[F_i\right] \\ &= \sum \frac{v_i^2 \sigma_i^2}{n_i}. \end{aligned}$$

If we make the reasonable assumption that the number of samples n_i is proportional to the volume v_i, then we have $n_i = v_i N$, and the variance of the overall estimator is

$$V\left[F_N\right] = \frac{1}{N} \sum v_i \sigma_i^2.$$

To compare this result to the variance without stratification, we note that choosing an unstratified sample is equivalent to choosing a random stratum I according to the discrete probability distribution defined by the volumes v_i, and then choosing a random sample X in Λ_I. In this sense, X is chosen *conditionally* on I, so it can be shown using



(a) (b)

Figure 14.2: Variance is higher and the image noisier (a) when random sampling is used to compute the effect of glossy reflection than (b) when a stratified distribution of sample directions is used instead. (Compare the edges of the highlights on the ground, for example.)

conditional probability that

$$\begin{aligned} V[F] &= E_x V_i F + V_x E_i F \\ &= \frac{1}{N}\left[\sum v_i \sigma_i^2 + \sum v_i \left(\mu_i - Q\right)\right], \end{aligned}$$

where Q is the mean of f over the whole domain Λ. See Veach (1997) for a derivation of this result.

There are two things to notice about this expression. First, we know that the right-hand sum must be nonnegative, since variance is always nonnegative. Second, it demonstrates that stratified sampling can never increase variance. In fact, stratification always reduces variance unless the right-hand sum is exactly zero. It can only be zero when the function f has the same mean over each stratum Λ_i. In fact, for stratified sampling to work best, we would like to maximize the right-hand sum, so it is best to make the strata have means that are as unequal as possible. This explains why *compact* strata are desirable if one does not know anything about the function f. If the strata are wide, they will contain more variation and will have μ_i closer to the true mean Q.

Figure 14.2 shows the effect of using stratified sampling versus a uniform random distribution for sampling ray directions for glossy reflection. There is a reasonable reduction in variance at essentially no cost in running time.

The main downside of stratified sampling is that it suffers from the same "curse of dimensionality" as standard numeric quadrature. Full stratification in D dimensions with S strata per dimension requires S^D samples, which quickly becomes prohibitive. Fortunately, it is often possible to stratify some of the dimensions independently and then randomly associate samples from different dimensions, as was done in Section 7.3.

Choosing which dimensions are stratified should be done in a way that stratifies dimensions that tend to be most highly correlated in their effect on the value of the integrand (Owen 1998). For example, for the direct lighting example in Section 14.1.1, it is far more effective to stratify the (x, y) pixel positions and to stratify the (θ, ϕ) ray direction—stratifying (x, θ) and (y, ϕ) would almost certainly be ineffective.

Another solution to the curse of dimensionality that has many of the same advantages of stratification is to use Latin hypercube sampling (also introduced in Section 7.3), which can generate any number of samples independent of the number of dimensions. Unfortunately, Latin hypercube sampling isn't as effective as stratified sampling at reducing variance, especially as the number of samples taken becomes large. Nevertheless, Latin hypercube sampling is provably no worse than uniform random sampling and is often much better.

14.2.2 QUASI MONTE CARLO

The low-discrepancy sampling techniques introduced in Section 7.4 are the foundation of a branch of Monte Carlo called *quasi Monte Carlo*. The key component of quasi–Monte Carlo techniques is that they replace the random numbers used in standard Monte Carlo with low-discrepancy point sets generated by carefully designed deterministic algorithms.

The advantage of this approach is that for many integration problems, quasi-Monte Carlo techniques have asymptotically faster rates of convergence than methods based on standard Monte Carlo. Many of the techniques used in regular Monte Carlo algorithms can be shown to work equally well with quasi-random sample points, including importance sampling. Some others (e.g., Russian roulette and rejection sampling) cannot. While the asymptotic convergence rates are not generally applicable to the discontinuous integrands in graphics because the convergence rates depend on smoothness properties in the integrand, quasi Monte Carlo generally performs better than regular Monte Carlo for these integrals in practice. The "Further Reading" section at the end of this chapter has more information about this topic.

In `pbrt`, we have generally glossed over the differences between these two approaches and have localized them in the `Sampler`s in Chapter 7. This introduces the possibility of subtle errors if a `Sampler` generates quasi-random sample points that an `Integrator` then improperly uses as part of an implementation of an algorithm that is not suitable for quasi Monte Carlo. As long as `Integrator`s only use these sample points for importance sampling or other techniques that are applicable in both approaches, this isn't a problem.

14.2.3 WARPING SAMPLES AND DISTORTION

When applying stratified sampling or low-discrepancy sampling to problems like choosing points on light sources for integration for area lighting, `pbrt` generates a set of samples (ξ_1, ξ_2) over the domain $[0, 1]^2$ and then uses algorithms based on the transformation methods introduced in Sections 13.5 and 13.6 to map these samples to points on the light source. Implicit in this process is the expectation that the transformation to points on the light source will generally preserve the stratification properties of the samples from $[0, 1]^2$—in other words, nearby samples should map to nearby positions on the surface

Integrator 740
Sampler 340

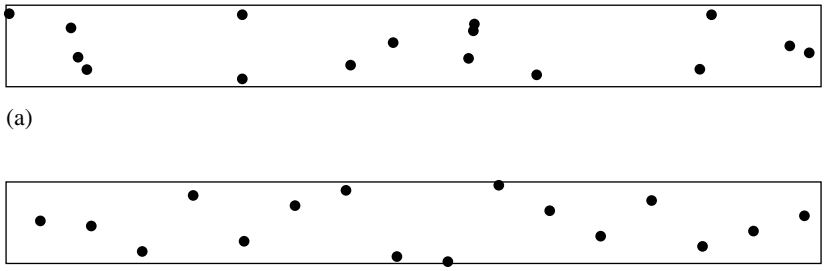


Figure 14.3: (a) Transforming a 4×4 stratified sampling pattern to points on a long and thin quadrilateral light source effectively gives less than 16 well-distributed samples; stratification in the vertical direction is not helpful. (b) Samples from a $(0, 2)$-sequence remain well distributed even after this transformation.

of the light, and faraway samples should map to far-apart positions on the light. If the mapping does not preserve this property, then the benefits of stratification are lost.

For example, this explains why Shirley's square-to-circle mapping (Figure 13.13) is better than the straightforward mapping (Figure 13.12), since the straightforward mapping has less compact strata away from the center. It also explains why $(0, 2)$-sequences can be so much better than stratified patterns in practice: they are more robust with respect to preserving their good distribution properties after being transformed to other domains. For example, Figure 14.3 shows what happens when a set of 16 well-distributed sample points are transformed to be points on a skinny quadrilateral by scaling them to cover its surface; the $(0, 2)$-sequence remains well-distributed, but the stratified pattern fares much less well.

14.3 BIAS

Another approach to variance reduction is to introduce *bias* into the computation: sometimes knowingly computing an estimate that doesn't actually have an expected value equal to the desired quantity can nonetheless lead to lower variance. An estimator is *unbiased* if its expected value is equal to the correct answer. If not, the difference

$$\beta = E[F] - \int f(x)\, \mathrm{d}x$$

is the amount of bias.

Kalos and Whitlock (1986, pp. 36–37) gave the following example of how bias can sometimes be desirable. Consider the problem of computing an estimate of the mean value of a distribution $X_i \sim p$ over the interval from zero to one. One could use the estimator

$$\frac{1}{N} \sum_{i=1}^{N} X_i,$$

or one could use the biased estimator

$$\frac{1}{2}\max(X_1, X_2, \ldots, X_N).$$

The first estimator is in fact unbiased, but has variance with order $O(N^{-1})$. The second estimator's expected value is

$$0.5\frac{N}{N+1} \neq 0.5,$$

so it is biased, although its variance is $O(N^{-2})$, which is much better. For large values of N, the second estimator may be preferred.

The pixel reconstruction method described in Section 7.7 can also be thought of as a biased estimator. Considering it as a Monte Carlo estimation problem, we'd like to compute an estimate of

$$I(x, y) = \iint f(x - x', y - y')L(x', y')\,\mathrm{d}x'\,\mathrm{d}y',$$

where $I(x, y)$ is a final pixel value, $f(x, y)$ is the pixel filter function (which we assume here to be normalized to integrate to one), and $L(x, y)$ is the image radiance function.

Assuming we have chosen image plane samples uniformly, all samples have the same probability density, which we will denote by p_c. Thus, the unbiased Monte Carlo estimator of this equation is

$$I(x, y) \approx \frac{1}{Np_c}\sum_{i=1}^{N} f(x - x_i, y - y_i)L(x_i, y_i).$$

This gives a different result than the pixel filtering equation we used previously, Equation (7.4), which was

$$I(x, y) = \frac{\sum_i f(x - x_i, y - y_i)L(x_i, y_i)}{\sum_i f(x - x_i, y - y_i)}.$$

Yet, the biased estimator is preferable in practice because it gives a result with less variance. For example, if all radiance values $L(x_i, y_i)$ have a value of one, the biased estimator will always reconstruct an image where all pixel values are exactly one—clearly a desirable property. However, the unbiased estimator will reconstruct pixel values that are not all one, since the sum

$$\sum_i f(x - x_i, y - y_i)$$

will generally not be equal to p_c and thus will have a different value due to variation in the filter function depending on the particular (x_i, y_i) sample positions used for the pixel. Thus, the variance due to this effect leads to an undesirable result in the final image. Even for more complex images, the variance that would be introduced by the unbiased estimator is a more objectionable artifact than the bias from Equation (7.4).

14.4 IMPORTANCE SAMPLING

Importance sampling is a variance reduction technique that exploits the fact that the Monte Carlo estimator

$$F_N = \frac{1}{N} \sum_{i=1}^{N} \frac{f(X_i)}{p(X_i)}$$

converges more quickly if the samples are taken from a distribution $p(x)$ that is similar to the function $f(x)$ in the integrand. The basic idea is that by concentrating work where the value of the integrand is relatively high, an accurate estimate is computed more efficiently (Figure 14.4).

For example, suppose we are evaluating the scattering equation, Equation (5.8). Consider what happens when this integral is estimated; if a direction is randomly sampled that is nearly perpendicular to the surface normal, the cosine term will be close to zero. All the expense of evaluating the BSDF and tracing a ray to find the incoming radiance at that sample location will be essentially wasted, as the contribution to the final result will be minuscule. As such, we would be better served if we sampled the sphere in a way that reduced the likelihood of choosing directions near the equator. More generally, if directions are sampled from distributions that match other factors of the integrand (the BSDF, the incoming illumination distribution, etc.), efficiency is similarly improved.

So long as the random variables are sampled from a probability distribution that is similar in shape to the integrand, variance is reduced. We will not provide a rigorous proof of this fact but will instead present an informal and intuitive argument. Suppose we're trying to use Monte Carlo techniques to evaluate some integral $\int f(x)\,\mathrm{d}x$. Since we have freedom in choosing a sampling distribution, consider the effect of using a distribution $p(x) \propto f(x)$, or $p(x) = cf(x)$. It is trivial to show that normalization forces

$$c = \frac{1}{\int f(x)\,\mathrm{d}x}.$$

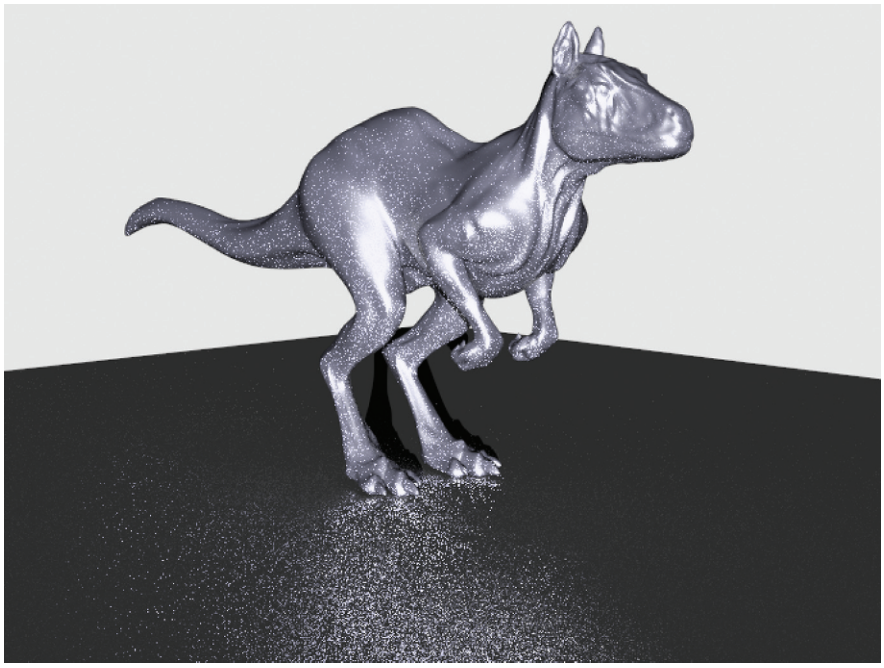
Finding such a PDF requires that we know the value of the integral, which is what we were trying to estimate in the first place. Nonetheless, for the purposes of this example, if we *could* sample from this distribution, each estimate would have the value

$$\frac{f(X_i)}{p(X_i)} = \frac{1}{c} = \int f(x)\,\mathrm{d}x.$$

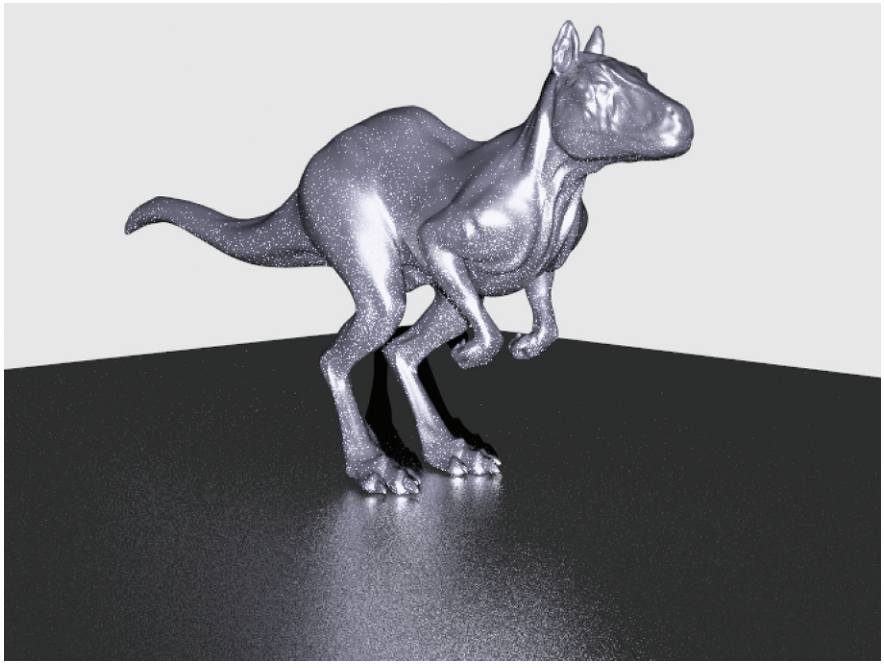
Since c is a constant, each estimate has the same value, and the variance is zero! Of course, this is ludicrous since we wouldn't bother using Monte Carlo if we could integrate f directly. However, if a density $p(x)$ can be found that is similar in shape to $f(x)$, variance decreases.

Importance sampling can increase variance if a poorly chosen distribution is used. To understand the effect of sampling from a PDF that is a poor match for the integrand, consider using the distribution

$$p(x) = \begin{cases} 99.01 & x \in [0, .01) \\ .01 & x \in [.01, 1) \end{cases}$$



(a)



(b)

Figure 14.4: (a) Using a stratified uniform distribution of rays over the hemisphere gives an image with much more variance than (b) applying importance sampling and choosing stratified rays from a distribution based on the BRDF.

to compute the estimate of $\int f(x)\,\mathrm{d}x$ where

$$f(x) = \begin{cases} .01 & x \in [0, .01) \\ 1.01 & x \in [.01, 1). \end{cases}$$

By construction, the value of the integral is one, yet using $p(x)$ to draw samples to compute a Monte Carlo estimate will give a terrible result: almost all of the samples will be in the range $[0, .01)$, where the estimator has the value $f(x)/p(x) \approx 0.0001$. For any estimate where none of the samples ends up being outside of this range, the result will be very inaccurate, almost 10,000 times smaller than it should be. Even worse is the case where some samples do end up being taken in the range $[.01, 1)$. This will happen rarely, but when it does, we have the combination of a relatively high value of the integrand and a relatively low value of the PDF, $f(x)/p(x) = 101$. A large number of samples would be necessary to balance out these extremes to reduce variance enough to get a result close to the actual value, one.

Fortunately, it's not too hard to find good sampling distributions for importance sampling for many integration problems in graphics. For example, in many cases, the integrand is the product of more than one function. It can be difficult to construct a PDF that is similar to the complete product, but finding one that is similar to one of the multiplicands can be helpful. This will be a common strategy in the light transport algorithms in the following chapters, where we will be trying to estimate integrals that multiply lighting, visibility, scattering, and cosine terms. (The "Further Reading" section at the end of this chapter has pointers to methods that generate PDFs that sample products of functions like these.)

In practice, importance sampling is one of the most frequently used variance reduction techniques in rendering, since it is easy to apply and is very effective when good sampling distributions are used. It is one of the variance reduction techniques of choice in `pbrt`, and therefore a variety of techniques for sampling from distributions from BSDFs, light sources, and functions related to participating media will be derived in this chapter.

14.4.1 MULTIPLE IMPORTANCE SAMPLING

Monte Carlo provides tools to estimate integrals of the form $\int f(x)\,\mathrm{d}x$. However, we are frequently faced with integrals that are the product of two or more functions: $\int f(x)g(x)\,\mathrm{d}x$. If we have an importance sampling strategy for $f(x)$ *and* a strategy for $g(x)$, which should we use? (Assume that we are not able to combine the two sampling strategies to compute a PDF that is proportional to the product $f(x)g(x)$ that can itself be sampled easily.) As shown in the discussion of importance sampling, a bad choice of sampling distribution can be much worse than just using a uniform distribution.

For example, consider the problem of evaluating direct lighting integrals of the form

$$L_\mathrm{o}(\mathrm{p}, \omega_\mathrm{o}) = \int_{\mathbb{S}^2} f(\mathrm{p}, \omega_\mathrm{o}, \omega_\mathrm{i})\, L_\mathrm{d}(\mathrm{p}, \omega_\mathrm{i})\, |\cos\theta_\mathrm{i}|\, \mathrm{d}\omega_\mathrm{i}.$$

If we were to perform importance sampling to estimate this integral according to distributions based on either L_d or f_r, one of these two will often perform poorly.

Consider a near-mirror BRDF illuminated by an area light where L_{d}'s distribution is used to draw samples. Because the BRDF is almost a mirror, the value of the integrand will be close to zero at all ω_{i} directions except those around the perfect specular reflection direction. This means that almost all of the directions sampled by L_{d} will have zero contribution, and variance will be quite high. Even worse, as the light source grows large and a larger set of directions is potentially sampled, the value of the PDF decreases, so for the rare directions where the BRDF is nonzero for the sampled direction we will have a large integrand being divided by a small PDF value. While sampling from the BRDF's distribution would be a much better approach to this particular case, for diffuse or glossy BRDFs and small light sources, sampling from the BRDF's distribution can similarly lead to much higher variance than sampling from the light's distribution.

Unfortunately, the obvious solution of taking some samples from each distribution and averaging the two estimators is hardly any better. Because the variance is additive in this case, this approach doesn't help—once variance has crept into an estimator, we can't eliminate it by adding it to another estimator even if it itself has low variance.

Multiple importance sampling (MIS) addresses exactly these kinds of problems, with a simple and easy-to-implement technique. The basic idea is that, when estimating an integral, we should draw samples from multiple sampling distributions, chosen in the hope that at least one of them will match the shape of the integrand reasonably well, even if we don't know which one this will be. MIS provides a method to weight the samples from each technique that can eliminate large variance spikes due to mismatches between the integrand's value and the sampling density. Specialized sampling routines that only account for unusual special cases are even encouraged, as they reduce variance when those cases occur, with relatively little cost in general. Figure 14.5 shows the reduction in variance from using MIS to compute reflection from direct illumination compared to the two approaches described previously.

If two sampling distributions p_f and p_g are used to estimate the value of $\int f(x)g(x)\,\mathrm{d}x$, the new Monte Carlo estimator given by MIS is

$$\frac{1}{n_f}\sum_{i=1}^{n_f}\frac{f(X_i)g(X_i)w_f(X_i)}{p_f(X_i)}+\frac{1}{n_g}\sum_{j=1}^{n_g}\frac{f(Y_j)g(Y_j)w_g(Y_j)}{p_g(Y_j)},$$

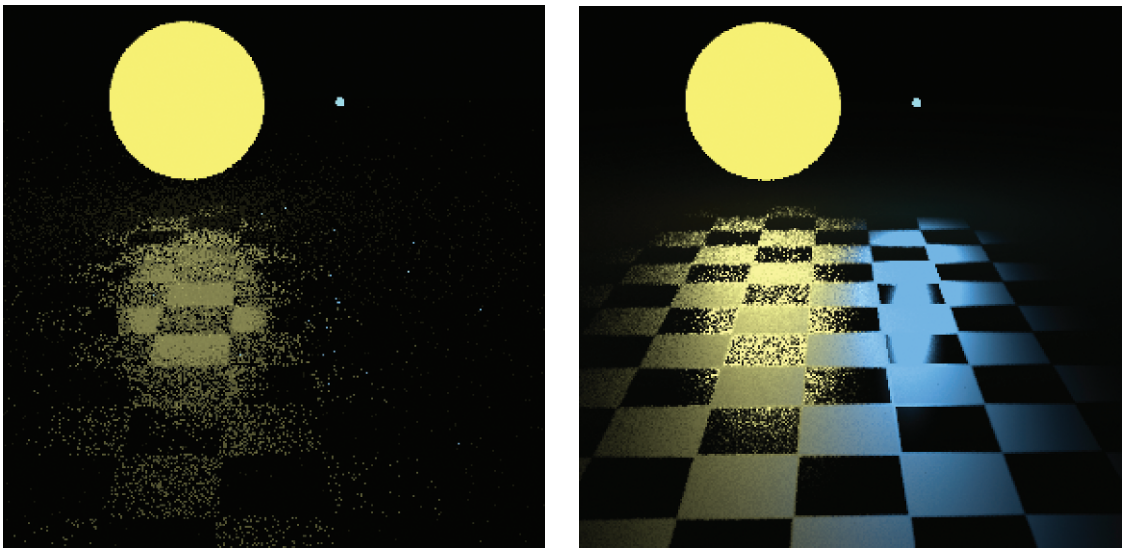
where n_f is the number of samples taken from the p_f distribution method, n_g is the number of samples taken from p_g, and w_f and w_g are special weighting functions chosen such that the expected value of this estimator is the value of the integral of $f(x)g(x)$.

The weighting functions take into account *all* of the different ways that a sample X_i or Y_j could have been generated, rather than just the particular one that was actually used. A good choice for this weighting function is the *balance heuristic*:

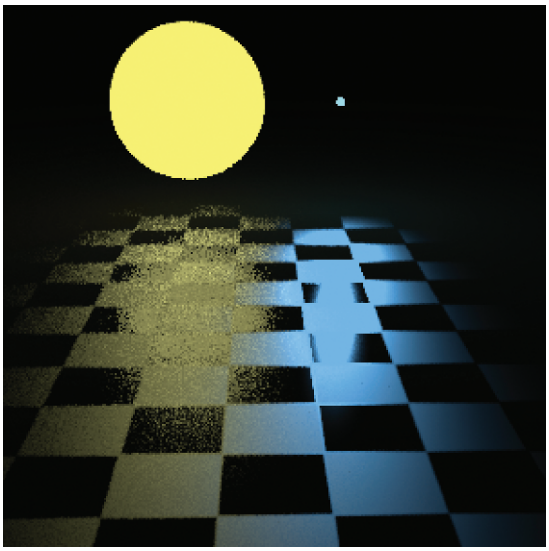
$$w_s(x)=\frac{n_s p_s(x)}{\sum_i n_i p_i(x)}.$$

The balance heuristic is a provably good way to weight samples to reduce variance.

Consider the effect of this term for the case where a sample X has been drawn from the p_f distribution at a point where the value $p_f(X)$ is relatively low. Assuming that p_f is



(a) (b)



(c)

Figure 14.5: Multiple importance sampling is a very effective modification to standard importance sampling methods: (a) A scene with a glossy checkerboard where some checks are nearly perfectly specular and others are nearly diffuse, where direct lighting is computed by sampling the BSDF. This strategy works well for the nearly specular checks but not for the diffuse ones. (b) The result of sampling from the light sources' distributions instead. Again, some reflections look good while others suffer from very high variance. (c) The result of using multiple importance sampling with the power heuristic, giving the best overall result.

a good match for the shape of $f(x)$, then the value of $f(X)$ will also be relatively low. But suppose that $g(X)$ has a relatively high value. The standard importance sampling estimate

$$\frac{f(X)g(X)}{p_f(X)}$$

will have a very large value due to $p_f(X)$ being small, and we will have high variance.

With the balance heuristic, the contribution of X will be

$$\frac{f(X)g(X)w_f(X)}{p_f(X)} = \frac{f(X)g(X)\, n_f\, p_f(X)}{p_f(X)(n_f\, p_f(X) + n_g\, p_g(X))} = \frac{f(X)g(X)\, n_f}{n_f\, p_f(X) + n_g\, p_g(X)}.$$

As long as p_g's distribution is a reasonable match for $g(x)$, then the denominator won't be too small thanks to the $n_g p_g(X)$ term, and the huge variance spike is eliminated, even though X was sampled from a distribution that was in fact a poor match for the integrand. The fact that another distribution will also be used to generate samples and that this new distribution will likely find a large value of the integrand at X are brought together in the weighting term to reduce the variance problem.

Here we provide an implementation of the balance heuristic for the specific case of two distributions p_f and p_g. We will not need a more general multidistribution case in `pbrt`.

⟨*Monte Carlo Inline Functions*⟩ ≡
```
inline float BalanceHeuristic(int nf, float fPdf, int ng, float gPdf) {
    return (nf * fPdf) / (nf * fPdf + ng * gPdf);
}
```

In practice, the *power heuristic* often reduces variance even further. For an exponent β, the power heuristic is

$$w_s(x) = \frac{(n_s p_s(x))^\beta}{\sum_i (n_i p_i(x))^\beta}.$$

Veach determined empirically that $\beta = 2$ is a good value. We have $\beta = 2$ hard-coded into the implementation here.

⟨*Monte Carlo Inline Functions*⟩ +≡
```
inline float PowerHeuristic(int nf, float fPdf, int ng, float gPdf) {
    float f = nf * fPdf, g = ng * gPdf;
    return (f*f) / (f*f + g*g);
}
```

14.5 SAMPLING REFLECTION FUNCTIONS

Because importance sampling is such an effective variance reduction technique, it's worth going through the effort to find sampling distributions that match functions in the common integrands in graphics. In this section, we will show the derivations and implementations for sampling strategies corresponding to distributions that are similar to many of the BSDF models used in `pbrt`.

The BxDF::Sample_f() method randomly chooses a direction according to a distribution that is similar to its corresponding scattering function. In Section 8.2, this method was used for finding reflected and transmitted rays from perfectly specular surfaces; later in this section, we will show how that sampling process is a special case of the sampling techniques of the previous chapter. BxDF::Sample_f() takes two sample values in the range $[0, 1)^2$ that are intended to be used by a transformation-based sampling algorithm. Since these are parameters to the method, the routine calling it can easily use stratified or low-discrepancy sampling techniques to generate them, thus ensuring well-distributed directions. If the algorithm used by BxDF::Sample_f() to generate directions doesn't need these sample values (e.g., if it uses rejection sampling), it can just ignore these arguments, although stratified sampling of a distribution that only roughly matches the BSDF is generally a more effective approach than rejection sampling from its distribution exactly.

This method returns the sampled direction in *wi and returns the value of $p(\omega_i)$ in *pdf. The value of the BSDF for the chosen direction is returned with a Spectrum return value. The PDF value returned should be measured with respect to solid angle on the hemisphere, and both the outgoing direction ω_o and the sampled incident direction ω_i should be in the standard reflection coordinate system (see "Geometric Setting," page 425).

The default implementation of this method samples the unit hemisphere with a cosine-weighted distribution. Samples from this distribution will give correct results for any BRDF that isn't described by a delta distribution, since there is some probability of sampling all directions where the BRDF's value is nonzero: $p(\omega) > 0$ for all ω. (BTDFs will thus always need to override this method but can sample the opposite hemisphere uniformly if they don't have a better sampling method.)

⟨*BxDF Method Definitions*⟩ +≡
```
Spectrum BxDF::Sample_f(const Vector &wo, Vector *wi,
                        float u1, float u2, float *pdf) const {
    ⟨Cosine-sample the hemisphere, flipping the direction if necessary 694⟩
    *pdf = Pdf(wo, *wi);
    return f(wo, *wi);
}
```

There is a subtlety related to the orientation of the normal in the reflection coordinate system that must be accounted for here: the direction returned by Malley's method will always be in the hemisphere around (0, 0, 1). If the ω_o direction is in the opposite hemisphere, then ω_i must be flipped to lie in the same hemisphere as ω_o. This issue is a direct consequence of the fact that pbrt does not flip the normal to be on the same side of the surface as the ω_o direction.

⟨*Cosine-sample the hemisphere, flipping the direction if necessary*⟩ ≡ 694, 701
```
*wi = CosineSampleHemisphere(u1, u2);
if (wo.z < 0.) wi->z *= -1.f;
```

While BxDF::Sample_f() returns the value of the PDF for the direction it chose, the BxDF::Pdf() method returns the value of the PDF for an arbitrary given direction. This method is useful for multiple importance sampling, where it is necessary to be able to find the PDF for directions sampled from other distributions. It is crucial that any

BxDF 428
BxDF::f() 429
BxDF::Pdf() 695
BxDF::Sample_f() 694
Spectrum 263
Vector 57

BxDF subclass that overrides the BxDF::Sample_f() method also override the BxDF::Pdf() method so that the two return consistent results.

To actually evaluate the PDF for the cosine-weighted sampling method (which we showed earlier was $p(\omega) = \cos\theta/\pi$), it is first necessary to check that ω_o and ω_i lie on the same side of the surface; if not, the sampling probability is zero. Otherwise, the method computes $|\mathbf{n} \cdot \omega_i|$. One potential pitfall with this method is that the order of the ω_o and ω_i arguments is significant. For the cosine-weighted distribution here, for example, $p(\omega_o) \neq p(\omega_i)$ in general. Code that calls this method must be careful to use the correct argument ordering.

⟨*BSDF Inline Functions*⟩ +≡
```
inline bool SameHemisphere(const Vector &w, const Vector &wp) {
    return w.z * wp.z > 0.f;
}
```

⟨*BxDF Method Definitions*⟩ +≡
```
float BxDF::Pdf(const Vector &wo, const Vector &wi) const {
    return SameHemisphere(wo, wi) ? AbsCosTheta(wi) * INV_PI : 0.f;
}
```

This sampling method works well for Lambertian BRDFs, and it works well for the Oren–Nayar model as well, so we will not override it for those classes.

14.5.1 SAMPLING THE BLINN MICROFACET DISTRIBUTION

BRDFs based on microfacet distribution functions are more difficult to sample. For these models, the BRDF is a product of three terms, D, G, and F, which is then divided by two cosine terms; recall Equation (8.8). It is impractical to find the probability distribution that matches the complete model, so instead we will derive a method for drawing samples from the distribution described by the microfacet distribution function D alone. This is an effective strategy for importance sampling the complete model, since the D term accounts for most of its variation.

Therefore, all MicrofacetDistribution implementations must implement methods for sampling from their distribution and computing the value of their PDF, each with the same signature as the corresponding BxDF function, except that their Sample_f() methods return void.

AbsCosTheta() 426
BxDF 428
BxDF::Pdf() 695
BxDF::Sample_f() 694
INV_PI 1002
Microfacet 454
MicrofacetDistribution 454
SameHemisphere() 695
Vector 57

⟨*MicrofacetDistribution Interface*⟩ +≡ 454
```
virtual void Sample_f(const Vector &wo, Vector *wi,
                      float u1, float u2, float *pdf) const = 0;
virtual float Pdf(const Vector &wo, const Vector &wi) const = 0;
```

The Microfacet BxDF, then, forwards the sampling and PDF requests to the distribution function that it holds a pointer to.

⟨*BxDF Method Definitions*⟩ +≡
```
Spectrum Microfacet::Sample_f(const Vector &wo, Vector *wi,
                              float u1, float u2, float *pdf) const {
    distribution->Sample_f(wo, wi, u1, u2, pdf);
    if (!SameHemisphere(wo, *wi)) return Spectrum(0.f);
    return f(wo, *wi);
}
```

⟨*BxDF Method Definitions*⟩ +≡
```
float Microfacet::Pdf(const Vector &wo, const Vector &wi) const {
    if (!SameHemisphere(wo, wi)) return 0.f;
    return distribution->Pdf(wo, wi);
}
```

Recall that Blinn's microfacet distribution function is $D(\cos\theta_h) = (n+2)(\cos\theta_h)^n$, where $\cos\theta_\mathrm{h} = |\mathbf{n}\cdot\omega_\mathrm{h}|$ and the distribution in terms of directions on the unit sphere is $D(\omega_\mathrm{h}) = ((n+2)/2\pi)(\cos\theta_\mathrm{h})^n$ (Equation (8.9)). Thus, in order to sample incident directions, we will first sample a half-angle direction ω_h according to this distribution, which we will then use to find ω_i by reflecting ω_o about the sampled direction ω_h.

⟨*BxDF Method Definitions*⟩ +≡
```
void Blinn::Sample_f(const Vector &wo, Vector *wi, float u1, float u2,
                     float *pdf) const {
    ⟨Compute sampled half-angle vector ωh for Blinn distribution 697⟩
    ⟨Compute incident direction by reflecting about ωh 697⟩
    ⟨Compute PDF for ωi from Blinn distribution 699⟩
    *pdf = blinn_pdf;
}
```

Because the value of ϕ doesn't affect D, the PDF for this distribution, $p_\mathrm{h}(\theta,\phi)$, is separable into $p_\mathrm{h}(\theta)$ and $p_\mathrm{h}(\phi)$. $p_\mathrm{h}(\phi)$ is constant with a value of $1/(2\pi)$, and thus ϕ values can be sampled by

$$\phi = 2\pi\xi_2.$$

By construction, microfacet distribution functions are already normalized. However, recall that the normalization in Section 8.4.3 was with respect to a projected solid angle, in order to ensure that the area of the microfacets was physically-plausible. Here, we want a normalized distribution of directions on the hemisphere. Following the usual approach, this can be shown to be

$$p_\mathrm{h}(\cos\theta_\mathrm{h}) = (n+1)\cos^n\theta_\mathrm{h}. \qquad [14.1]$$

This is a power distribution in $\cos\theta_\mathrm{h}$, and thus a uniform random variable can be used to generate a sample by

$$\cos\theta_\mathrm{h} = \sqrt[n+1]{\xi_1}.$$

(Recall Equation (13.4).)

Blinn 456
Microfacet 454
Microfacet::f() 455
MicrofacetDistribution::Pdf() 695
MicrofacetDistribution::Sample_f() 695
SameHemisphere() 695
Spectrum 263
Vector 57

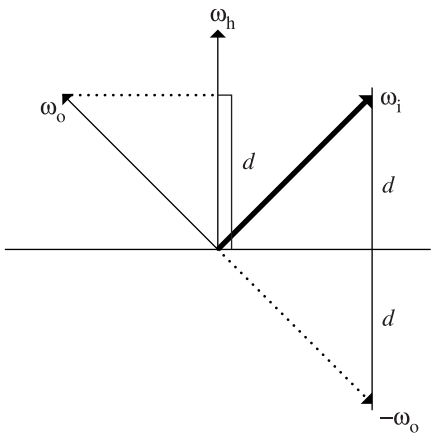


Figure 14.6: The reflection of a direction ω_o about the direction ω_h can be computed by first taking the offset $-\omega_o$ from the origin, giving the vector beneath the surface. We then add two times the distance d, which is given by the projection of ω_o onto ω_h (which is given by their dot product) to give the direction ω_i above the surface.

Because `pbrt` transforms the normal to (0, 0, 1) in the reflection coordinate system, we can almost use the computed direction from spherical coordinates directly. The only additional detail to handle is that if ω_o is in the opposite hemisphere than the normal, then the half-angle vector needs to be flipped to be in that hemisphere as well.

⟨*Compute sampled half-angle vector ω_h for Blinn distribution*⟩ ≡ 696
```
float costheta = powf(u1, 1.f / (exponent+1));
float sintheta = sqrtf(max(0.f, 1.f - costheta*costheta));
float phi = u2 * 2.f * M_PI;
Vector wh = SphericalDirection(sintheta, costheta, phi);
if (!SameHemisphere(wo, wh)) wh = -wh;
```

All that's left to do is to apply the formula for reflection of a vector about another vector (Figure 14.6):

⟨*Compute incident direction by reflecting about ω_h*⟩ ≡ 696, 699
```
*wi = -wo + 2.f * Dot(wo, wh) * wh;
```

There's an important detail to take care of to compute the value of the PDF for the sampled direction, however. The microfacet distribution gives the distribution of normals around the *half-angle vector*, but the reflection integral is with respect to the *incoming vector*. These distributions are not the same, and we must convert the half-angle PDF to the incoming angle PDF. In other words, we must change from a density in terms of ω_h to one in terms of ω_i using the techniques introduced in Section 13.5. Doing so requires applying the adjustment for a change of variables $d\omega_h/d\omega_i$.

M_PI 1002
SameHemisphere() 695
SphericalDirection() 292
Vector 57

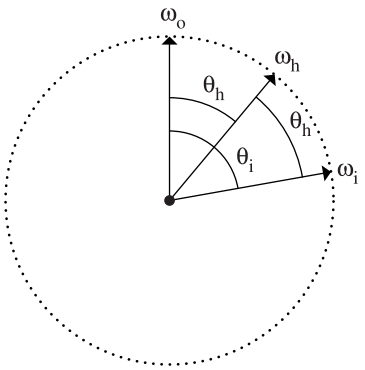


Figure 14.7: The adjustment for change of variable from sampling from the half-angle distribution to sampling from the incident direction distribution can be derived with an observation about the relative angles involved: $\theta_i = 2\theta_h$.

A simple geometric construction gives the relationship between the two distributions. Consider the spherical coordinate system oriented about ω_o (Figure 14.7). The differential solid angles $d\omega_i$ and $d\omega_h$ are $\sin\theta_i\, d\theta_i d\phi_i$ and $\sin\theta_h\, d\theta_h d\phi_h$, respectively; thus,

$$\frac{d\omega_h}{d\omega_i} = \frac{\sin\theta_h\, d\theta_h\, d\phi_h}{\sin\theta_i\, d\theta_i\, d\phi_i}.$$

Because ω_i is computed by reflecting ω_o about ω_h, $\theta_i = 2\theta_h$. Furthermore, because $\phi_i = \phi_h$, we can find the desired conversion factor:

$$\begin{aligned}\frac{d\omega_h}{d\omega_i} &= \frac{\sin\theta_h\, d\theta_h\, d\phi_h}{\sin 2\theta_h\, 2\, d\theta_h\, d\phi_h} \\ &= \frac{\sin\theta_h}{4\cos\theta_h \sin\theta_h} \\ &= \frac{1}{4\cos\theta_h} \\ &= \frac{1}{4(\omega_i \cdot \omega_h)} = \frac{1}{4(\omega_o \cdot \omega_h)}.\end{aligned}$$

Therefore, the PDF after transformation is

$$p(\theta) = \frac{p_h(\theta)}{4(\omega_o \cdot \omega_h)}.$$

Using the definition of the Blinn distribution over the hemisphere from Equation (14.1), we have:

⟨*Compute PDF for ω_i from Blinn distribution*⟩ ≡ 696, 699
```
float blinn_pdf = ((exponent + 1.f) * powf(costheta, exponent)) /
                  (2.f * M_PI * 4.f * Dot(wo, wh));
if (Dot(wo, wh) <= 0.f) blinn_pdf = 0.f;
```

We can reuse this fragment to implement the `Blinn::Pdf()` method.

⟨*BxDF Method Definitions*⟩ +≡
```
float Blinn::Pdf(const Vector &wo, const Vector &wi) const {
    Vector wh = Normalize(wo + wi);
    float costheta = AbsCosTheta(wh);
    ⟨Compute PDF for ωi from Blinn distribution 699⟩
    return blinn_pdf;
}
```

14.5.2 SAMPLING THE ANISOTROPIC MICROFACET MODEL

Ashikhmin and Shirley (2000, 2002) gave the equations for sampling from the distribution defined by their anisotropic BRDF model. We will restate their results here without deriving them; interested readers should consult the original publications for details and derivations.

⟨*BxDF Method Definitions*⟩ +≡
```
void Anisotropic::Sample_f(const Vector &wo, Vector *wi,
                           float u1, float u2, float *pdf) const {
    ⟨Sample from first quadrant and remap to hemisphere to sample ωh 699⟩
    ⟨Compute incident direction by reflecting about ωh 697⟩
    ⟨Compute PDF for ωi from anisotropic distribution 701⟩
    *pdf = anisotropic_pdf;
}
```

The sampling method in the `Anisotropic::sampleFirstQuadrant()` function samples a direction in the first quadrant of the unit hemisphere—that is, with spherical angles in the range $(\theta, \phi) \in [0, \pi/2] \times [0, \pi/2]$. It's easier to derive a sampling method for this distribution by considering only that part of the problem and using it as a building block for sampling all directions. Here, we use ξ_1 to select a quadrant, remap it to $[0, 1]$ to generate a sample from the first quadrant, and then map the sample to the desired quadrant. The remapping of ξ_1 and the sampled ϕ value are all done carefully to preserve stratification: given a small difference in ξ_1 that moves from one quadrant to another, the generated half-angle direction should change only slightly.

⟨*Sample from first quadrant and remap to hemisphere to sample ω_h*⟩ ≡ 699
```
float phi, costheta;
if (u1 < .25f) {
    sampleFirstQuadrant(4.f * u1, u2, &phi, &costheta);
} else if (u1 < .5f) {
    u1 = 4.f * (.5f - u1);
    sampleFirstQuadrant(u1, u2, &phi, &costheta);
    phi = M_PI - phi;
```

AbsCosTheta() 426
Anisotropic 458
Anisotropic::sampleFirstQuadrant() 700
Blinn 456
Blinn::exponent 456
Blinn::Pdf() 699
Dot() 60
M_PI 1002
SameHemisphere() 695
SphericalDirection() 292
Vector 57
Vector::Normalize() 63

```
} else if (u1 < .75f) {
    u1 = 4.f * (u1 - .5f);
    sampleFirstQuadrant(u1, u2, &phi, &costheta);
    phi += M_PI;
} else {
    u1 = 4.f * (1.f - u1);
    sampleFirstQuadrant(u1, u2, &phi, &costheta);
    phi = 2.f * M_PI - phi;
}
float sintheta = sqrtf(max(0.f, 1.f - costheta*costheta));
Vector wh = SphericalDirection(sintheta, costheta, phi);
if (!SameHemisphere(wo, wh)) wh = -wh;
```

The half-angle vector in the first quadrant of the hemisphere can be sampled from this distribution function by

$$\phi = \arctan\left(\sqrt{\frac{e_x+1}{e_y+1}}\tan\left(\frac{\pi\xi_1}{2}\right)\right)$$

$$\cos\theta = \xi_2^{(e_x\cos^2\phi + e_y\sin^2\phi+1)^{-1}}.$$

Note that, if $e_x = e_y$ and the microfacet distribution is isotropic, this gives the same sampling technique as was used for the earlier Blinn distribution.

⟨*BxDF Method Definitions*⟩ +≡

```
void Anisotropic::sampleFirstQuadrant(float u1, float u2,
        float *phi, float *costheta) const {
    if (ex == ey)
        *phi = M_PI * u1 * 0.5f;
    else
        *phi = atanf(sqrtf((ex+1.f) / (ey+1.f)) *
                     tanf(M_PI * u1 * 0.5f));
    float cosphi = cosf(*phi), sinphi = sinf(*phi);
    *costheta = powf(u2, 1.f/(ex * cosphi * cosphi +
                              ey * sinphi * sinphi + 1));
}
```

Similar to the Blinn case, the PDF for a direction sampled with this method is the value of the normalized distribution over the hemisphere of directions, although as before we must account for the change of variables required to convert from the half-angle distribution to the incident angle distribution just as was done in `Blinn::Pdf()`.

⟨*BxDF Method Definitions*⟩ +≡

```
float Anisotropic::Pdf(const Vector &wo, const Vector &wi) const {
    Vector wh = Normalize(wo + wi);
    ⟨Compute PDF for ωi from anisotropic distribution 701⟩
    return anisotropic_pdf;
}
```

Anisotropic 458
Anisotropic::ex 458
Anisotropic::ey 458
Blinn::Pdf() 699
M_PI 1002
Vector 57
Vector::Normalize() 63

⟨*Compute PDF for ω_i from anisotropic distribution*⟩ ≡ 699, 700
```
float costhetah = AbsCosTheta(wh);
float ds = 1.f - costhetah * costhetah;
float anisotropic_pdf = 0.f;
if (ds > 0.f && Dot(wo, wh) > 0.f) {
    float e = (ex * wh.x * wh.x + ey * wh.y * wh.y) / ds;
    float d = sqrtf((ex+1.f) * (ey+1.f)) * INV_TWOPI *
              powf(costhetah, e);
    anisotropic_pdf = d / (4.f * Dot(wo, wh));
}
```

14.5.3 SAMPLING FresnelBlend

The FresnelBlend class is a mixture of a diffuse and glossy term. A straightforward approach to sampling this BRDF is to sample from both a cosine-weighted distribution as well as the microfacet distribution. The implementation here chooses between the two with equal probability based on whether ξ_1 is less than or greater than 0.5. In both cases, it remaps ξ_1 to cover the range $[0, 1]$ after using it to make this decision. (Otherwise, values of ξ_1 used for the cosine-weighted sampling would always be less than 0.5, for example.) Using the sample ξ_1 for two purposes in this manner slightly reduces the quality of the stratification of the (ξ_1, ξ_2) values that are actually used for sampling directions, although this is less of a shortcoming than randomly selecting between the two sampling methods with a new unstratified random number.

⟨*BxDF Method Definitions*⟩ +≡
```
Spectrum FresnelBlend::Sample_f(const Vector &wo, Vector *wi,
                                float u1, float u2, float *pdf) const {
    if (u1 < .5) {
        u1 = 2.f * u1;
        ⟨Cosine-sample the hemisphere, flipping the direction if necessary 694⟩
    }
    else {
        u1 = 2.f * (u1 - .5f);
        distribution->Sample_f(wo, wi, u1, u2, pdf);
        if (!SameHemisphere(wo, *wi)) return Spectrum(0.f);
    }
    *pdf = Pdf(wo, *wi);
    return f(wo, *wi);
}
```

The PDF for this sampling strategy is simple; it is just an average of the two PDFs used.

⟨*BxDF Method Definitions*⟩ +≡
```
float FresnelBlend::Pdf(const Vector &wo, const Vector &wi) const {
    if (!SameHemisphere(wo, wi)) return 0.f;
    return .5f * (AbsCosTheta(wi) * INV_PI + distribution->Pdf(wo, wi));
}
```

AbsCosTheta() 426
FresnelBlend 460
INV_PI 1002
INV_TWOPI 1002
MicrofacetDistribution::Pdf() 695
MicrofacetDistribution::Sample_f() 695
Spectrum 263
Vector 57

14.5.4 SPECULAR REFLECTION AND TRANSMISSION

The Dirac delta distributions that were previously used to define the BRDF for specular reflection and the BTDF for specular transmission fit into this sampling framework well, as long as a few conventions are kept in mind when using their sampling and PDF functions.

Recall that the Dirac delta distribution is defined such that

$$\delta(x) = 0 \text{ for all } x \neq 0$$

and

$$\int_{-\infty}^{\infty} \delta(x)\, \mathrm{d}x = 1.$$

Thus, it is a probability density function, where the PDF has a value of zero for all $x \neq 0$. Generating a sample from such a distribution is trivial; there is only one possible value for it to take on. When thought of in this way, the implementations of `Sample_f()` for the `SpecularReflection` and `SpecularTransmission` `BxDF`s can be seen to fit naturally into the Monte Carlo sampling framework.

It is not as simple to determine which value should be returned for the value of the PDF. Strictly speaking, the delta distribution is not a true function, but must be defined as the limit of another function—for example, one describing a box of unit area whose width approaches zero; see Chapter 5 of Bracewell (2000) for further discussion and references. Thought of in this way, the value of $\delta(0)$ tends toward infinity. Certainly, returning an infinite or very large value for the PDF is not going to lead to correct results from the renderer.

Recall that BSDFs defined with delta components also have these delta components in their f_r functions, a detail that was glossed over when we returned values from their `Sample_f()` methods in Chapter 8. Thus, the Monte Carlo estimator for the scattering equation with such a BSDF is written

$$\frac{1}{N}\sum_{i}^{N} \frac{f_\mathrm{r}(\mathrm{p}, \omega_\mathrm{o}, \omega_i) L_\mathrm{i}(\mathrm{p}, \omega_i) |\cos\theta_i|}{p(\omega_i)} = \frac{1}{N}\sum_{i}^{N} \frac{\rho_\mathrm{hd}(\omega_\mathrm{o}) \frac{\delta(\omega - \omega_i)}{|\cos\theta_i|} L_\mathrm{i}(\mathrm{p}, \omega_i) |\cos\theta_i|}{p(\omega_i)},$$

where $\rho_\mathrm{hd}(\omega_\mathrm{o})$ is the hemispherical–directional reflectance and ω is the direction for perfect specular reflection or transmission.

Because the PDF $p(\omega_i)$ has a delta term as well, $p(\omega_i) = \delta(\omega - \omega_i)$, the two delta distributions cancel out, and the estimator is

$$\rho_\mathrm{hd}(\omega_\mathrm{o}) L_\mathrm{i}(\mathrm{p}, \omega),$$

exactly the quantity computed by the Whitted integrator, for example.

Therefore, the implementations here return a constant value of one for the PDF for specular reflection and transmission when sampled using `Sample_f()`, with the convention that for specular `BxDF`s there is an implied delta distribution in the PDF value that is expected to cancel out with the implied delta distribution in the value of the BSDF when the estimator is evaluated. The respective `Pdf()` methods therefore return zero for all di-

BxDF 428
SpecularReflection 440
SpecularTransmission 444

rections, since there is zero probability that another sampling method will randomly find the direction from a delta distribution.

⟨*SpecularReflection Public Methods*⟩ +≡ 440
```
float Pdf(const Vector &wo, const Vector &wi) const {
    return 0.;
}
```

⟨*SpecularTransmission Public Methods*⟩ +≡ 444
```
float Pdf(const Vector &wo, const Vector &wi) const {
    return 0.;
}
```

There is a potential pitfall with this convention: when multiple importance sampling is used to compute weights, PDF values that include these implicit delta distributions can't be freely mixed with regular PDF values. This isn't a problem in practice, since there's no reason to apply MIS when there's a delta distribution in the integrand. The light transport routines in the next chapter have appropriate logic to be sure to avoid this error.

14.5.5 APPLICATION: ESTIMATING REFLECTANCE

As an example of their application, we will show how these BxDF sampling routines can be used in computing estimates of the reflectance integrals defined in Section 8.1.1 for arbitrary BRDFs. For example, recall that the hemispherical–directional reflectance is

$$\rho_{\mathrm{hd}}(\omega_{\mathrm{o}}) = \int_{\mathcal{H}^2(\mathbf{n})} f_{\mathrm{r}}(\omega_{\mathrm{o}}, \omega_{\mathrm{i}}) \, |\cos \theta_{\mathrm{i}}| \, \mathrm{d}\omega_{\mathrm{i}}.$$

Recall from Section 8.1.1 that BxDF::rho() method implementations take two additional parameters, nSamples and samples; here, we can see how they are used for Monte Carlo sampling. For BxDF implementations that can't compute the reflectance in closed form, the nSamples parameter specifies the number of Monte Carlo samples to take, and sample values themselves are provided in the samples array. The samples array should have 2*nSamples entries, where the first two values provide the first 2D sample value and so forth.

The generic BxDF::rho() method computes a Monte Carlo estimate of this value for any BxDF, using the provided samples and taking advantage of the BxDF's sampling method to compute the estimate with importance sampling.

⟨*BxDF Method Definitions*⟩ +≡
```
Spectrum BxDF::rho(const Vector &w, int nSamples,
                   const float *samples) const {
    Spectrum r = 0.;
    for (int i = 0; i < nSamples; ++i) {
        ⟨Estimate one term of ρhd 704⟩
    }
    return r / float(nSamples);
}
```

BxDF 428
BxDF::rho() 430
Spectrum 263
Vector 57

Actually evaluating the estimator is a matter of sampling the reflection function's distribution, finding its value, and dividing it by the value of the PDF. Each term of the estimator

$$\frac{1}{N}\sum_{j}^{N}\frac{f_{\mathrm{r}}(\omega,\omega_j)\,|\cos\theta_j|}{p(\omega_j)}$$

is easily evaluated. The BxDF's Sample_f() method returns all of ω_j, $p(\omega_j)$ and the value of $f_{\mathrm{r}}(\omega_{\mathrm{o}},\omega_j)$. The only tricky part is when $p(\omega_j)=0$, which must be detected here, since otherwise a division by zero would place an infinite value in r.

⟨*Estimate one term of* ρ_{hd}⟩ ≡ 703
```
Vector wi;
float pdf = 0.f;
Spectrum f = Sample_f(w, &wi, samples[2*i], samples[2*i+1], &pdf);
if (pdf > 0.) r += f * AbsCosTheta(wi) / pdf;
```

The hemispherical–hemispherical reflectance can be estimated similarly. Given

$$\rho_{\mathrm{hh}}=\frac{1}{\pi}\int_{\mathcal{H}^2(\mathbf{n})}\int_{\mathcal{H}^2(\mathbf{n})}f_{\mathrm{r}}(\omega',\omega'')\,|\cos\theta'\cos\theta''|\,\mathrm{d}\omega'\,\mathrm{d}\omega'',$$

two vectors, ω' and ω'', must be sampled for each term of the estimate

$$\frac{1}{N}\sum_{j}^{N}\frac{f_{\mathrm{r}}(\omega'_j,\omega''_j)\,|\cos\theta'_j\cos\theta''_j|}{p(\omega'_j)\,p(\omega''_j)}.$$

⟨*BxDF Method Definitions*⟩ +≡
```
Spectrum BxDF::rho(int nSamples, const float *samples1,
                   const float *samples2) const {
    Spectrum r = 0.;
    for (int i = 0; i < nSamples; ++i) {
        ⟨Estimate one term of ρhh 704⟩
    }
    return r / (M_PI*nSamples);
}
```

The implementation here samples the first direction ω' uniformly over the hemisphere. Given this, the second direction can be sampled with the BxDF::Sample_f() method.[1]

⟨*Estimate one term of* ρ_{hh}⟩ ≡ 704
```
Vector wo, wi;
wo = UniformSampleHemisphere(samples1[2*i], samples1[2*i+1]);
float pdf_o = INV_TWOPI, pdf_i = 0.f;
Spectrum f = Sample_f(wo, &wi, samples2[2*i], samples2[2*i+1], &pdf_i);
if (pdf_i > 0.)
    r += f * AbsCosTheta(wi) * AbsCosTheta(wo) / (pdf_o * pdf_i);
```

AbsCosTheta() 426
BxDF 428
BxDF::Sample_f() 694
INV_TWOPI 1002
M_PI 1002
Spectrum 263
UniformSampleHemisphere() 664
Vector 57

1 It could be argued that a shortcoming of the BxDF sampling interface is that there aren't entry points to sample from the four-dimensional distribution of $f_{\mathrm{r}}(\mathrm{p},\omega,\omega')$. This is a reasonably esoteric case for the applications envisioned for pbrt, however.

14.5.6 SAMPLING BSDFs

Given these methods to sample individual BxDFs, we can now define a sampling method for the BSDF class, BSDF::Sample_f(). This method is called by SurfaceIntegrators when they want to sample according to the BSDF's distribution; it calls the individual BxDF::Sample_f() methods to generate samples. The BSDF stores pointers to one or more individual BxDFs that can be sampled individually, but here we will sample from the density that is the sum of their individual densities,

$$p(\omega) = \frac{1}{N} \sum_{i}^{N} p_i(\omega).$$

(Exercise 14.1 at the end of the chapter discusses the alternative of sampling from the BxDFs according to probabilities based on their respective reflectances; this approach can be more efficient if they vary significantly.)

The BSDF::Sample_f() method takes three random variables to drive this process; one is used to select among the BxDFs, and the other two to pass along to the particular sampling method chosen. These sample values are encapsulated by the BSDFSample structure.

⟨*BSDF Declarations*⟩ +≡
```
struct BSDFSample {
    ⟨BSDFSample Public Methods 705⟩
    float uDir[2], uComponent;
};
```

The BSDFSample can be initialized by directly passing in values for the three samples.

⟨*BSDFSample Public Methods*⟩ ≡ 705
```
BSDFSample(float up0, float up1, float ucomp) {
    uDir[0] = up0;
    uDir[1] = up1;
    uComponent = ucomp;
}
```

Alternatively, a convenience method takes a pseudo-random number generator and initializes the samples using it.

⟨*BSDFSample Public Methods*⟩ +≡ 705
```
BSDFSample(RNG &rng) {
    uDir[0] = rng.RandomFloat();
    uDir[1] = rng.RandomFloat();
    uComponent = rng.RandomFloat();
}
```

Most commonly, we'll want to initialize the BSDFSample with values from a Sample object filled in by the Sampler. The BSDFSampleOffsets structure helps with the bookkeeping for this case.

BSDF 478
BSDF::Sample_f() 706
BSDFSample 705
BSDFSample::uComponent 705
BSDFSample::uDir 705
BSDFSampleOffsets 706
BxDF 428
BxDF::Sample_f() 694
RNG 1003
RNG::RandomFloat() 1003
Sample 343
Sampler 340
SurfaceIntegrator 740

⟨*BSDF Declarations*⟩ +≡
```
struct BSDFSampleOffsets {
    int nSamples, componentOffset, dirOffset;
};
```

Its constructor takes the total number of samples to be taken and the `Sample` object from the `Sampler`; it calls the `Add1D()` method to request 1D samples for choosing a `BxDF` component, and the `Add2D()` method to request 2D samples for the sampled direction.

⟨*BSDF Method Definitions*⟩ +≡
```
BSDFSampleOffsets::BSDFSampleOffsets(int count, Sample *sample) {
    nSamples = count;
    componentOffset = sample->Add1D(nSamples);
    dirOffset = sample->Add2D(nSamples);
}
```

With the `BSDFSampleOffsets` structure in hand, we can implement a third constructor for the `BSDFSample` class. It takes a filled-in `Sample`, a previously initialized `BSDFSampleOffsets`, and the sample number to be taken. (`num` should be less than the `count` value passed to the `BSDFSampleOffsets` constructor). It then takes care of extracting the appropriate values from the `Sample`'s sample values and initializing the member variables.

⟨*BSDF Method Definitions*⟩ +≡
```
BSDFSample::BSDFSample(const Sample *sample,
                       const BSDFSampleOffsets &offsets, uint32_t n) {
    uDir[0] = sample->twoD[offsets.dirOffset][2*n];
    uDir[1] = sample->twoD[offsets.dirOffset][2*n+1];
    uComponent = sample->oneD[offsets.componentOffset][n];
}
```

With the definition of `BSDFSample` in hand, the `BSDF::Sample_f()` method can now be implemented. The outgoing direction passed to it and the incident direction it returns are in world space, which is indicated by the `W` at the end of the respective variable names.

⟨*BSDF Method Definitions*⟩ +≡
```
Spectrum BSDF::Sample_f(const Vector &woW, Vector *wiW,
                        const BSDFSample &bsdfSample, float *pdf,
                        BxDFType flags, BxDFType *sampledType) const {
    ⟨Choose which BxDF to sample 707⟩
    ⟨Sample chosen BxDF 707⟩
    ⟨Compute overall PDF with all matching BxDFs 707⟩
    ⟨Compute value of BSDF for sampled direction 708⟩
}
```

This method first determines which `BxDF`'s sampling method to use for this particular sample. This is complicated by the fact that the caller may pass in flags that the chosen `BxDF` must match (e.g., specifying that only diffuse components should be considered). Thus, only a subset of the sampling densities may actually be used here.

BSDF 478
BSDFSample 705
BSDFSample::uComponent 705
BSDFSample::uDir 705
BSDFSampleOffsets 706
BSDFSampleOffsets::componentOffset 706
BSDFSampleOffsets::dirOffset 706
BxDF 428
BxDFType 428
Sample 343
Sample::Add1D() 344
Sample::Add2D() 344
Sample::oneD 344
Sample::twoD 344
Sampler 340
Spectrum 263
Vector 57

⟨*Choose which* BxDF *to sample*⟩ ≡ 706
```
int matchingComps = NumComponents(flags);
if (matchingComps == 0) {
    *pdf = 0.f;
    return Spectrum(0.f);
}
int which = min(Floor2Int(bsdfSample.uComponent * matchingComps),
                matchingComps-1);
BxDF *bxdf = NULL;
int count = which;
for (int i = 0; i < nBxDFs; ++i)
    if (bxdfs[i]->MatchesFlags(flags) && count-- == 0) {
        bxdf = bxdfs[i];
        break;
    }
```

Once the appropriate BxDF is chosen, its BxDF::Sample_f() method is called. Recall that these methods expect and return vectors in the BxDF's local coordinate system, so the supplied vector must be transformed to the BxDF's coordinate system and the returned vector must be transformed back into world coordinates.

⟨*Sample chosen* BxDF⟩ ≡ 706
```
Vector wo = WorldToLocal(woW);
Vector wi;
*pdf = 0.f;
Spectrum f = bxdf->Sample_f(wo, &wi, bsdfSample.uDir[0],
                            bsdfSample.uDir[1], pdf);
if (*pdf == 0.f)
    return 0.f;
if (sampledType) *sampledType = bxdf->type;
*wiW = LocalToWorld(wi);
```

To compute the actual PDF for sampling the direction ω_i, we need the average of all of the PDFs of the BxDFs that could have been used, given the BxDFType flags passed in. Because *pdf already holds the PDF value for the distribution the sample was taken from, we only need to add in the contributions of the others. It's important that this step be skipped if the chosen BxDF is perfectly specular, since the PDF has an implicit delta distribution in it. It would be incorrect to add the other PDF values to this one, since it is a delta term represented with value one, rather than as an actual delta distribution.

⟨*Compute overall PDF with all matching* BxDFs⟩ ≡ 706
```
if (!(bxdf->type & BSDF_SPECULAR) && matchingComps > 1)
    for (int i = 0; i < nBxDFs; ++i)
        if (bxdfs[i] != bxdf && bxdfs[i]->MatchesFlags(flags))
            *pdf += bxdfs[i]->Pdf(wo, wi);
if (matchingComps > 1) *pdf /= matchingComps;
```

Given the sampled direction, this method needs to compute the value of the BSDF for the pair of directions (ω_o, ω_i) accounting for all of the relevant components in the

BSDF::bxdfs 479
BSDF::LocalToWorld() 480
BSDF::NumComponents() 479
BSDF::WorldToLocal() 480
BSDF_SPECULAR 428
BxDF 428
BxDF::MatchesFlags() 429
BxDF::Pdf() 695
BxDF::Sample_f() 694
BxDF::type 429
BxDFType 428
Floor2Int() 1002
Spectrum 263
Vector 57

BSDF, unless the sampled direction was from a specular component, in which case the value returned from `Sample_f()` earlier is used. (If a specular component generated this direction, its `BxDF::f()` method will return black, even if we pass back the direction its sampling routine returned.)

While this method could just call the `BSDF::f()` method to compute the BSDF's value, the value can be more efficiently computed by calling the `BxDF::f()` methods directly, taking advantage of the fact that here we already have the directions in both world space and the reflection coordinate system available.

⟨*Compute value of BSDF for sampled direction*⟩ ≡ 706
```
if (!(bxdf->type & BSDF_SPECULAR)) {
    f = 0.;
    if (Dot(*wiW, ng) * Dot(woW, ng) > 0) // ignore BTDFs
        flags = BxDFType(flags & ~BSDF_TRANSMISSION);
    else // ignore BRDFs
        flags = BxDFType(flags & ~BSDF_REFLECTION);
    for (int i = 0; i < nBxDFs; ++i)
        if (bxdfs[i]->MatchesFlags(flags))
            f += bxdfs[i]->f(wo, wi);
}
return f;
```

The `BSDF::Pdf()` method does a similar computation, looping over the `BxDF`s and calling their `Pdf()` methods to find the PDF for an arbitrary sampled direction:

⟨*BSDF Public Methods*⟩ +≡ 478
```
float Pdf(const Vector &wo, const Vector &wi,
          BxDFType flags = BSDF_ALL) const;
```

14.6 SAMPLING LIGHT SOURCES

Because direct illumination from light sources makes a key contribution to reflected light at a point, it is important to be able to take a point and sample those directions around it where the direct illumination may take on a nonzero value. Consider a diffuse surface illuminated by a small spherical area light source (Figure 14.8): sampling directions using the BSDF's sampling distribution is likely to be very inefficient because the light is only visible along a small cone of directions from the point.

A much better approach is to instead use a sampling distribution that is based on the light source. For example, the sampling routine could choose from among only those directions where the sphere is potentially visible. The sphere may actually be occluded for some or all of these directions, though it is difficult to account for this issue in defining the sampling densities.

This section will apply the now familiar Monte Carlo machinery to the light sampling problem. Furthermore, it will show the derivations and implementations of techniques for sampling rays leaving light sources. Being able to do this is crucial for bidirectional light transport algorithms like photon mapping and bidirectional path tracing.

BSDF::f() 481
BSDF::Pdf() 708
BSDF_ALL 428
BSDF_REFLECTION 428
BSDF_SPECULAR 428
BSDF_TRANSMISSION 428
BxDF 428
BxDF::f() 429
BxDF::MatchesFlags() 429
BxDFType 428
Vector 57

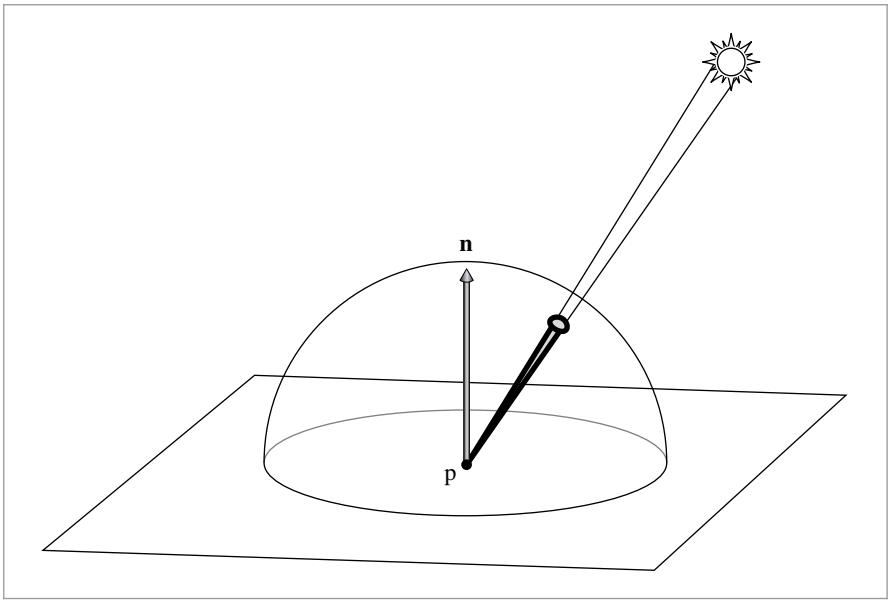


Figure 14.8: Another effective sampling strategy for choosing an incident direction from a point p for direct lighting computations is to use the light source to define a distribution of directions with respect to solid angle at the point. Here, a small spherical light source is illuminating the point. The cone of directions that the sphere subtends is a much better sampling distribution to use than a uniform distribution, for example.

A number of the light transport algorithms in the following chapters need to randomly sample a light source based on its power; for efficiency, it's generally worthwhile to take more samples from light sources that have a greater effect on the illumination in the scene. Sampling lights by their emitted power is generally a good proxy for this. The `ComputeLightSamplingCDF()` utility function creates a `Distribution1D` based on the power of all of the lights in the scene to compute a cumulative distribution function for sampling from the lights.

⟨*Integrator Utility Functions*⟩ +≡

```
Distribution1D *ComputeLightSamplingCDF(const Scene *scene) {
    uint32_t nLights = int(scene->lights.size());
    vector<float>lightPower(nLights, 0.f);
    for (uint32_t i = 0; i < nLights; ++i)
        lightPower[i] = scene->lights[i]->Power(scene).y();
    return new Distribution1D(&lightPower[0], nLights);
}
```

BxDF::Pdf() 695
BxDF::Sample_f() 694
Distribution1D 648
Light 606
Light::Pdf() 711
Light::Power() 608
Light::Sample_L() 608
Scene 22
Scene::lights 23
Spectrum::y() 273

14.6.1 BASIC INTERFACE

There are three main methods related to sampling that all `Lights` must implement. Two of them, `Light::Sample_L()` and `Light::Pdf()`, are comparable to the `BxDF::Sample_f()` and `BxDF::Pdf()` methods; they also generate samples from a distribution of directions

over the sphere from a point in the scene. Here, the distribution doesn't try to match the BSDF at the point, but instead tries to match the incident radiance distribution from direct illumination from the light source.

The `Light::Sample_L()` method was first introduced in Section 12.1. For reference, here is its declaration:

```
virtual Spectrum Sample_L(const Point &p, float pEpsilon,
    const LightSample &ls, float time, Vector *wi, float *pdf,
    VisibilityTester *vis) const = 0;
```

We can now understand the meaning of its `LightSample` and `pdf` parameters: `LightSample` stores a three-dimensional random sample for sampling the light source (similar to the `BSDFSample`), and the PDF for sampling the chosen direction is returned in `*pdf`. Not all lights need all three of these samples, but for lights that are comprised of many `Shapes`, then we need one sample to select which shape to sample from and two samples to choose a point on its surface. The definition of `LightSample` is almost the same as `BSDFSample`, so we'll elide much of its definition here.

⟨*Light Declarations*⟩ +≡
```
struct LightSample {
    ⟨LightSample Public Methods⟩
    float uPos[2], uComponent;
};
```

Also similar to `BSDFSampleOffsets`, `LightSampleOffsets` handles requesting appropriate sample values from the `Sampler` and then extracting them from a specific `Sample`.

⟨*Light Declarations*⟩ +≡
```
struct LightSampleOffsets {
    LightSampleOffsets(int count, Sample *sample);
    int nSamples, componentOffset, posOffset;
};
```

⟨*Light Method Definitions*⟩ +≡
```
LightSampleOffsets::LightSampleOffsets(int count, Sample *sample) {
    nSamples = count;
    componentOffset = sample->Add1D(nSamples);
    posOffset = sample->Add2D(nSamples);
}
```

⟨*Light Method Definitions*⟩ +≡
```
LightSample::LightSample(const Sample *sample,
        const LightSampleOffsets &offsets, uint32_t n) {
    uPos[0] = sample->twoD[offsets.posOffset][2*n];
    uPos[1] = sample->twoD[offsets.posOffset][2*n+1];
    uComponent = sample->oneD[offsets.componentOffset][n];
}
```

Code that calls `Light::Sample_L()` should also check the value returned by the light's `IsDeltaLight()` method to see if the light source is described by a delta distribution (e.g.,

BSDFSample 705
BSDFSampleOffsets 706
Light::Sample_L() 608
LightSample 710
LightSample::uComponent 710
LightSample::uPos 710
LightSampleOffsets 710
LightSampleOffsets::
componentOffset 710
LightSampleOffsets::
posOffset 710
Sample 343
Sample::Add1D() 344
Sample::Add2D() 344
Sample::oneD 344
Sample::twoD 344
Sampler 340

a point light). There are similar issues related to how to handle PDF values returned from lights described by delta distributions as there are for perfectly specular BSDFs (recall the discussion in Section 14.5.4).

The `Light`'s `Pdf()` method returns the probability density with respect to solid angle for the light's `Sample_L()` method sampling the direction `wi` from the point `p`.

⟨*Light Interface*⟩ +≡ 606
```
virtual float Pdf(const Point &p, const Vector &wi) const = 0;
```

The third light sampling method is quite different. It samples a ray from a distribution of rays *leaving* the light, returning the ray in `*ray` and the surface normal at the point on the light source in `*Ns`. Being able to sample rays in this manner is necessary to support algorithms that construct paths of light leaving the light source.

The returned PDF value should be expressed in terms of the product of the density with respect to surface area on the light and the density with respect to solid angle. The implementations of this method in this section will sample the two distributions separately and compute their product, rather than trying to construct and sample a joint four-dimensional distribution. Implementations of this method may need up to five sample values, with the two additional ones used to determine the direction of the ray (after the first three are used to determine its origin on the light source).

⟨*Light Interface*⟩ +≡ 606
```
virtual Spectrum Sample_L(const Scene *scene, const LightSample &ls,
                          float u1, float u2, float time, Ray *ray,
                          Normal *Ns, float *pdf) const = 0;
```

14.6.2 LIGHTS WITH SINGULARITIES

Just as with perfect specular reflection and transmission, light sources that are defined in terms of delta functions fit naturally into this sampling framework, although they require care on the part of the routines that call their sampling methods, since there are implicit delta distributions in the radiance and PDF values that they return. For the most part, these delta distributions naturally cancel out when estimators are evaluated, although multiple importance sampling code must be aware of this case, just as was the case with BSDFs.

`Light` 606
`LightSample` 710
`Normal` 65
`Point` 63
`PointLight::Sample_L()` 611
`Ray` 66
`Scene` 22
`Spectrum` 263
`Vector` 57

Point Lights

Point lights are described by a delta distribution such that they only illuminate a receiving point from a single direction. Thus, the sampling problem is trivial. The `PointLight::Sample_L()` method was already implemented in Section 12.2, where we glossed over the nuance that the method performs Monte Carlo sampling of a delta distribution and thus always returns a single direction and doesn't need the random sample values.

Due to the delta distribution, the `PointLight::Pdf()` method returns zero. This reflects the fact that there is no chance for some other sampling process to randomly generate a point on an infinitesimal light source.

⟨*PointLight Method Definitions*⟩ +≡
```
float PointLight::Pdf(const Point &, const Vector &) const {
    return 0.;
}
```

The sampling method for generating rays leaving point lights is also straightforward. The origin of the ray must be the light's position; this part of the density is described with a delta distribution. Directions are uniformly sampled over the sphere, and the overall sampling density is the product of these two densities. As usual, we'll ignore the delta distribution that is in the actual PDF because it is canceled out by a (missing) corresponding delta term in the radiance value in the `Spectrum` returned by the sampling routine.

⟨*PointLight Method Definitions*⟩ +≡
```
Spectrum PointLight::Sample_L(const Scene *scene, const LightSample &ls,
        float u1, float u2, float time, Ray *ray, Normal *Ns,
        float *pdf) const {
    *ray = Ray(lightPos, UniformSampleSphere(ls.uPos[0], ls.uPos[1]),
               0.f, INFINITY, time);
    *Ns = (Normal)ray->d;
    *pdf = UniformSpherePdf();
    return Intensity;
}
```

Spotlights

The `SpotLight::Sample_L()` method was also previously implemented in Section 12.2.1; it also returns zero from its `Pdf()` method.

The method for sampling an outgoing ray with a reasonable distribution for the spotlight is more interesting. While it could just sample directions uniformly on the sphere as was done for the point light, this distribution is likely to be a bad match for the spotlight's actual distribution. For example, if the light has a very narrow beam angle, many samples will be taken in directions where the light doesn't cast any illumination. Instead, we will sample from a uniform distribution over the cone of directions in which the light casts illumination. Although the sampling distribution does not try to account for the falloff toward the edges of the beam, this is only a minor shortcoming in practice.

The PDF $p(\theta, \phi)$ for the spotlight's illumination distribution is separable with $p(\phi) = 1/(2\pi)$. We thus just need to find the distribution for θ and to sample from this distribution. We would like to sample a direction θ uniformly over the cone of directions around a central direction up to the maximum angle of the beam, $\theta_{\max}$. Incorporating the $\sin\theta$ term from the measure on the unit sphere from Equation (5.4),

$$\begin{aligned} 1 &= c \int_0^{\theta_{\max}} \sin\theta \, d\theta \\ &= c(1 - \cos\theta_{\max}). \end{aligned}$$

So $p(\theta) = \sin\theta(1 - \cos\theta_{\max})$ and $p(\omega) = 1/(2\pi(1 - \cos\theta_{\max}))$.

INFINITY 1002
LightSample 710
Normal 65
Point 63
PointLight 610
PointLight::Intensity 611
PointLight::lightPos 611
Ray 66
Scene 22
Spectrum 263
SpotLight::Sample_L() 713
UniformSampleSphere() 664
UniformSpherePdf() 664
Vector 57

⟨*Monte Carlo Function Definitions*⟩ +≡
```
float UniformConePdf(float cosThetaMax) {
    return 1.f / (2.f * M_PI * (1.f - cosThetaMax));
}
```

The PDF can be integrated to find the CDF, and the sampling technique,

$$\cos\theta = (1-\xi) + \xi\cos\theta_{\max},$$

follows. There are two `UniformSampleCone()` functions that implement this sampling technique: the first samples about the (0, 0, 1) axis, and the second (not shown here) takes three basis vectors for the coordinate system to be used where samples taken are with respect to the z axis of the given coordinate system.

⟨*Monte Carlo Function Definitions*⟩ +≡
```
Vector UniformSampleCone(float u1, float u2, float costhetamax) {
    float costheta = (1.f - u1) + u1 * costhetamax;
    float sintheta = sqrtf(1.f - costheta*costheta);
    float phi = u2 * 2.f * M_PI;
    return Vector(cosf(phi) * sintheta, sinf(phi) * sintheta, costheta);
}
```

The ray sampling method for the light uses these utility routines to choose a random ray inside the spotlight's cone.

⟨*SpotLight Method Definitions*⟩ +≡
```
Spectrum SpotLight::Sample_L(const Scene *scene, const LightSample &ls,
        float u1, float u2, float time, Ray *ray, Normal *Ns,
        float *pdf) const {
    Vector v = UniformSampleCone(ls.uPos[0], ls.uPos[1], cosTotalWidth);
    *ray = Ray(lightPos, LightToWorld(v), 0.f, INFINITY, time);
    *Ns = (Normal)ray->d;
    *pdf = UniformConePdf(cosTotalWidth);
    return Intensity * Falloff(ray->d);
}
```

GonioPhotometricLight 618
INFINITY 1002
Light::LightToWorld 606
LightSample 710
M_PI 1002
Normal 65
PointLight 610
ProjectionLight 614
ProjectionLight::
cosTotalWidth 617
Ray 66
Scene 22
Spectrum 263
SpotLight 612
SpotLight::Falloff() 614
SpotLight::Intensity 613
UniformConePdf() 713
UniformSampleCone() 713
Vector 57

Projection Lights and Goniophotometric Lights

The sampling routines for `ProjectionLights` and `GonioPhotometricLights` are essentially the same as `SpotLights` and `PointLights`, respectively. For sampling outgoing rays, `ProjectionLights` sample uniformly from the cone that encompasses their projected image map (hence the need to compute `ProjectionLight::cosTotalWidth` in the constructor), and those for `GonioPhotometricLights` sample uniformly over the unit sphere. An exercise at the end of this chapter discusses possible improvements to these sampling methods that better account for the directional variation of these lights.

Directional Lights

Sampling the incident illumination from a directional light at a given point is trivial. As with `PointLight` and its relatives, the `Sample_L()` method was implemented in Chapter 12 and its PDF method returns zero.

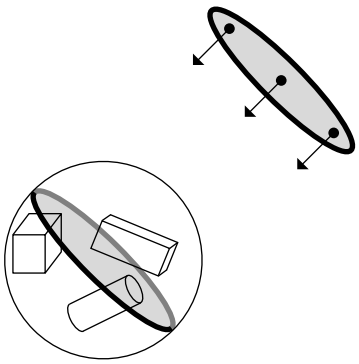


Figure 14.9: To sample an outgoing ray direction for a distant light source, the `DistantLight::Sample_L()` method finds the disk oriented in the light's direction that is large enough so that the entire scene can be intersected by rays leaving the disk in the light's direction. Ray origins are sampled uniformly by area on this disk, and ray directions are given directly by the light's direction.

Sampling a ray from the light's distribution of outgoing rays is a more interesting problem. The ray's direction is determined in advance by a delta distribution; it must be the same as the light's negated direction. For its origin, there are an infinite number of 3D points where it could start. How should we choose an appropriate one, and how do we compute its density?

The desired property is that rays intersect points in the scene that are illuminated by the distant light with uniform probability. One way to do this is to construct a disk that has the same radius as the scene's bounding sphere and has a normal that is oriented with the light's direction, and then choose a random point on this disk, using the `ConcentricSampleDisk()` function (Figure 14.9). Once this point has been chosen, if the point is displaced along the light's direction by the scene's bounding sphere radius and used as the origin of the light ray, the ray origin will be outside the bounding sphere of the scene, but will intersect it.

This is a valid sampling approach, since by construction it has nonzero probability of sampling all incident rays into the sphere due to the directional light. The area component of the sampling density is uniform and therefore equal to the reciprocal of the area of the disk that was sampled. The directional density is given by a delta distribution based on the light's direction; therefore, it isn't included in the returned PDF value.

⟨*DistantLight Method Definitions*⟩ +≡

```
Spectrum DistantLight::Sample_L(const Scene *scene,
        const LightSample &ls, float u1, float u2, float time,
        Ray *ray, Normal *Ns, float *pdf) const {
    ⟨Choose point on disk oriented toward infinite light direction 715⟩
    ⟨Set ray origin and direction for infinite light ray 715⟩
    *Ns = (Normal)ray->d;
```

ConcentricSampleDisk() 667
LightSample 710
M_PI 1002
Normal 65
Ray 66
Scene 22
Spectrum 263

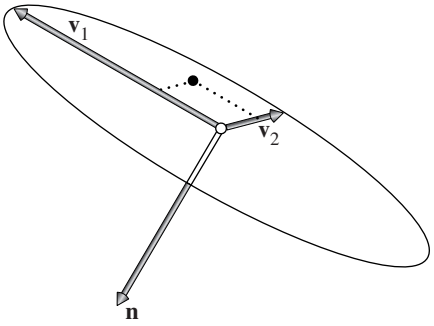


Figure 14.10: Given sample points (d_1, d_2) on the canonical unit disk, points on an arbitrarily oriented and sized disk with normal $\mathbf{n}$ can be found by computing an arbitrary coordinate system $(\mathbf{v}_1, \mathbf{v}_2, \mathbf{n})$, and then computing points on the disk with the offset $d_1\mathbf{v}_1 + d_2\mathbf{v}_2$ from the disk's center.

```
    *pdf = 1.f / (M_PI * worldRadius * worldRadius);
    return L;
}
```

Choosing the point on the oriented disk is a simple application of vector algebra. We construct a coordinate system with two vectors perpendicular to the disk's normal (the light's direction); see Figure 14.10. Given a random point on the canonical unit disk, computing the offsets from the disk's center with respect to its coordinate vectors gives the corresponding point.

⟨*Choose point on disk oriented toward infinite light direction*⟩ ≡ 714

```
Point worldCenter;
float worldRadius;
scene->WorldBound().BoundingSphere(&worldCenter, &worldRadius);
Vector v1, v2;
CoordinateSystem(lightDir, &v1, &v2);
float d1, d2;
ConcentricSampleDisk(ls.uPos[0], ls.uPos[1], &d1, &d2);
Point Pdisk = worldCenter + worldRadius * (d1 * v1 + d2 * v2);
```

And finally, the point is offset along the light direction and the ray is initialized:

⟨*Set ray origin and direction for infinite light ray*⟩ ≡ 714

```
*ray = Ray(Pdisk + worldRadius * lightDir, -lightDir, 0.f, INFINITY,
           time);
```

BBox::BoundingSphere() 74
ConcentricSampleDisk() 667
CoordinateSystem() 63
INFINITY 1002
Point 63
Ray 66
Scene::WorldBound() 24
Vector 57

14.6.3 AREA LIGHTS

Recall that area lights are defined by attaching an emission profile to a shape. Therefore, in order to sample incident illumination from such a light source, it is useful to be able to generate samples over the surface of the shape. To make this possible, we will add

sampling methods to the Shape class that sample random points on their surfaces.[2] The AreaLight sampling methods then call these methods.

Sampling Shapes

There are two shape sampling methods, both named Shape::Sample(). The first chooses points on the surface using some sampling distribution with respect to surface area on the shape and returns the position and surface normal of the point chosen.

⟨*Shape Interface*⟩ +≡ 108
```
virtual Point Sample(float u1, float u2, Normal *Ns) const {
    Severe("Unimplemented Shape::Sample() method called");
    return Point();
}
```

The Shapes that implement this method will almost always sample uniformly by area on their surface. Therefore, we will provide a default implementation of the Shape::Pdf() method corresponding to this sampling method that returns the corresponding PDF: the reciprocal of the surface area.

⟨*Shape Interface*⟩ +≡ 108
```
virtual float Pdf(const Point &Pshape) const {
    return 1.f / Area();
}
```

The second shape sampling method also takes the point from which the surface of the shape is being integrated over. This method is particularly useful for lighting, since the caller can pass in the point to be lit and allow shape implementations to ensure that they only sample the portion of the shape that is potentially visible from that point. The default implementation ignores the additional point and calls the earlier sampling method.

⟨*Shape Interface*⟩ +≡ 108
```
virtual Point Sample(const Point &P, float u1, float u2,
                     Normal *Ns) const {
    return Sample(u1, u2, Ns);
}
```

Unlike the first Shape sampling method, which generates points on the shape according to a probability density with respect to surface area on the shape, the second one uses a density with respect to solid angle from the point p. This difference stems from the fact that the area light sampling routines evaluate the direct lighting integral as an integral over directions from p—expressing these sampling densities with respect to solid angle at p is more convenient. Therefore, the standard implementation of the second Pdf()

AreaLight 623
Normal 65
Point 63
Sample 343
Severe() 1005
Shape 108
Shape::Area() 113
Shape::Pdf() 717
Shape::Sample() 716

2 Notice that the Shape::Sample() method is not a C++ "pure virtual" function. This is deliberate; not all shapes will implement a Sample() method. Some shapes are very difficult to sample (e.g., fractals), and it would be unreasonable to require those shapes to implement a Sample method just to be used as geometry in the system. Shapes like these should return false from their CanIntersect() method and should be able to refine themselves into shapes that do implement this method.

method here transforms the density from one defined over area to one defined over solid angle from p.

⟨*Shape Method Definitions*⟩ +≡
```
float Shape::Pdf(const Point &p, const Vector &wi) const {
    ⟨Intersect sample ray with area light geometry 717⟩
    ⟨Convert light sample weight to solid angle measure 717⟩
    return pdf;
}
```

Given a point p and direction ω_i, the `Pdf()` method determines if the ray (p, ω_i) intersects the shape. If the ray doesn't intersect the shape at all, the probability that the shape would have chosen the direction ω_i can be assumed to be zero (a well-written sampling algorithm wouldn't generate such a sample, and in any case the light's contribution will be zero for such directions, so using a zero probability density is fine).

Note that this ray intersection test is only between the ray and the single shape that is the area light source. The rest of the scene geometry is ignored, thus the intersection test is reasonably efficient.

⟨*Intersect sample ray with area light geometry*⟩ ≡ 717
```
DifferentialGeometry dgLight;
Ray ray(p, wi, 1e-3f);
float thit, rayEpsilon;
if (!Intersect(ray, &thit, &rayEpsilon, &dgLight)) return 0.;
```

To compute the value of the PDF with respect to solid angle from p, this method starts by computing the PDF with respect to surface area. Conversion from a density with respect to area to a density with respect to solid angle requires division by the factor

$$\frac{d\omega_i}{dA} = \frac{\cos\theta_o}{r^2},$$

where θ_o is the angle between the ray from the point on the light to the receiving point p and the light's surface normal, and r^2 is the distance between the point on the light and the point being shaded (recall Section 5.5).

⟨*Convert light sample weight to solid angle measure*⟩ ≡ 717
```
float pdf = DistanceSquared(p, ray(thit)) /
            (AbsDot(dgLight.nn, -wi) * Area());
```

AbsDot() 61
DifferentialGeometry 102
DiffuseAreaLight 625
DiffuseAreaLight::Sample_L() 718
DistanceSquared() 65
Point 63
Ray 66
Shape 108
Shape::Area() 113
Shape::Intersect() 111
Vector 57

Area Light Sampling Methods

Given shape sampling methods, the `DiffuseAreaLight::Sample_L()` methods are quite straightforward. Most of the hard work is done by the `Shapes`, and the `DiffuseAreaLight` mostly just needs to compute emitted radiance values. Recall that the shape or shapes that define the light are stored in a `ShapeSet` that handles sampling from collections of shapes. Its sampling routines will be defined in Section 14.6.4.

⟨*DiffuseAreaLight Method Definitions*⟩ +≡
```
Spectrum DiffuseAreaLight::Sample_L(const Point &p, float pEpsilon,
        const LightSample &ls, float time, Vector *wi, float *pdf,
        VisibilityTester *visibility) const {
    Normal ns;
    Point ps = shapeSet->Sample(p, ls, &ns);
    *wi = Normalize(ps - p);
    *pdf = shapeSet->Pdf(p, *wi);
    visibility->SetSegment(p, pEpsilon, ps, 1e-3f, time);
    Spectrum Ls = L(ps, ns, -*wi);
    return Ls;
}
```

The variant of `Shape::Pdf()` called here is the one that returns a density with respect to solid angle, so the value from the light's `Pdf()` method can be returned directly.

⟨*DiffuseAreaLight Method Definitions*⟩ +≡
```
float DiffuseAreaLight::Pdf(const Point &p, const Vector &wi) const {
    return shapeSet->Pdf(p, wi);
}
```

The method for sampling a ray leaving an area light is also easily implemented in terms of the shape sampling methods. The first variant of `Shape::Sample()` is used to find the ray origin, sampled from some density over the surface. Because the `DiffuseAreaLight` emits uniform radiance in all directions, a uniform distribution of directions is used for the ray direction. Because radiance is only emitted from the side of the light the surface normal lies on, if the sampled direction is in the opposite hemisphere than the surface normal, it is flipped to lie in the same hemisphere so that samples aren't wasted in directions where there is no emission.

The PDF for sampling the ray is the product of the PDF for sampling its origin with respect to area on the surface with the PDF for sampling a uniform direction on the hemisphere, $1/(2\pi)$.

⟨*DiffuseAreaLight Method Definitions*⟩ +≡
```
Spectrum DiffuseAreaLight::Sample_L(const Scene *scene,
        const LightSample &ls, float u1, float u2, float time,
        Ray *ray, Normal *Ns, float *pdf) const {
    Point org = shapeSet->Sample(ls, Ns);
    Vector dir = UniformSampleSphere(u1, u2);
    if (Dot(dir, *Ns) < 0.) dir *= -1.f;
    *ray = Ray(org, dir, 1e-3f, INFINITY, time);
    *pdf = shapeSet->Pdf(org) * INV_TWOPI;
    Spectrum Ls = L(org, *Ns, dir);
    return Ls;
}
```

Sampling Disks

The `Disk` sampling method uses the concentric disk sampling function to find a point on the unit disk and then scales and offsets this point to lie on the disk of a given

DiffuseAreaLight 625
DiffuseAreaLight::L() 626
Disk 129
Dot() 60
INFINITY 1002
INV_TWOPI 1002
LightSample 710
Normal 65
Point 63
Ray 66
Scene 22
Shape::Pdf() 717
Shape::Sample() 716
ShapeSet::Pdf() 724
ShapeSet::Sample() 723
Spectrum 263
UniformSampleSphere() 664
Vector 57
Vector::Normalize() 63
VisibilityTester 608
VisibilityTester::
SetSegment() 609

radius and height. Note that this method does not account for partial disks due to `Disk::innerRadius` being nonzero or `Disk::phiMax` being less than 2π. Solutions to this bug are discussed in an exercise at the end of the chapter.

⟨*Disk Method Definitions*⟩ +≡
```
Point Disk::Sample(float u1, float u2, Normal *Ns) const {
    Point p;
    ConcentricSampleDisk(u1, u2, &p.x, &p.y);
    p.x *= radius;
    p.y *= radius;
    p.z = height;
    *Ns = Normalize((*ObjectToWorld)(Normal(0,0,1)));
    if (ReverseOrientation) *Ns *= -1.f;
    return (*ObjectToWorld)(p);
}
```

Sampling Cylinders

Uniform sampling on cylinders is straightforward. The height and ϕ value are sampled uniformly. Intuitively, it can be understood that this works because a cylinder is just a rolled-up rectangle.

⟨*Cylinder Method Definitions*⟩ +≡
```
Point Cylinder::Sample(float u1, float u2, Normal *Ns) const {
    float z = Lerp(u1, zmin, zmax);
    float t = u2 * phiMax;
    Point p = Point(radius * cosf(t), radius * sinf(t), z);
    *Ns = Normalize((*ObjectToWorld)(Normal(p.x, p.y, 0.)));
    if (ReverseOrientation) *Ns *= -1.f;
    return (*ObjectToWorld)(p);
}
```

ConcentricSampleDisk() 667
Cross() 62
Cylinder::phiMax 126
Cylinder::radius 126
Cylinder::zmax 126
Cylinder::zmin 126
Disk::height 130
Disk::innerRadius 130
Disk::phiMax 130
Disk::radius 130
Lerp() 1000
Normal 65
Point 63
Sample 343
Shape::ObjectToWorld 108
Shape::
ReverseOrientation 108
Triangle 139
UniformSampleTriangle() 671
Vector::Normalize() 63

Sampling Triangles

The `UniformSampleTriangle()` function, defined in the previous chapter, returns the barycentric coordinates for a uniformly sampled point on a triangle. The point on a particular triangle for those barycentrics is easily computed.

⟨*TriangleMesh Method Definitions*⟩ +≡
```
Point Triangle::Sample(float u1, float u2, Normal *Ns) const {
    float b1, b2;
    UniformSampleTriangle(u1, u2, &b1, &b2);
    ⟨Get triangle vertices in p1, p2, and p3 140⟩
    Point p = b1 * p1 + b2 * p2 + (1.f - b1 - b2) * p3;
    Normal n = Normal(Cross(p2-p1, p3-p1));
    *Ns = Normalize(n);
    if (ReverseOrientation) *Ns *= -1.f;
    return p;
}
```

Sampling Spheres

As with `Disks`, the sampling method here does not handle partial spheres; an exercise at the end of the chapter discusses this issue further. If the sampling method is not given an external point that's being lit, sampling a point on a sphere is extremely simple. It just uses the `UniformSampleSphere()` function to generate a point on the unit sphere and scales the point by the sphere's radius.

⟨*Sphere Method Definitions*⟩ +≡

```
Point Sphere::Sample(float u1, float u2, Normal *ns) const {
    Point p = Point(0,0,0) + radius * UniformSampleSphere(u1, u2);
    *ns = Normalize((*ObjectToWorld)(Normal(p.x, p.y, p.z)));
    if (ReverseOrientation) *ns *= -1.f;
    return (*ObjectToWorld)(p);
}
```

For the sphere sampling method that is given a point being illuminated, however, it can do better. While uniform sampling over its surface leads to a correct estimate, a better approach is not to sample points on the sphere that are definitely not visible to the point being shaded (such as those on the back side of the sphere as seen from the point). The sampling routine will instead uniformly sample directions over the solid angle subtended by the sphere from the point being shaded. It generates directions inside this cone of directions by sampling an offset θ from the center vector ω_c and then sampling a rotation angle ϕ around the vector.

As seen from the point being shaded p the sphere subtends an angle of

$$\theta_{\max} = \arcsin\left(\frac{r}{|\mathrm{p} - \mathrm{p_c}|}\right) = \arccos\sqrt{1 - \left(\frac{r}{|\mathrm{p} - \mathrm{p_c}|}\right)^2}, \qquad [14.2]$$

where r is the radius of the sphere and $\mathrm{p_c}$ is its center (Figure 14.11). The implementation uniformly samples directions inside this cone and computes their intersection with the sphere to get the sample position on the light source.

⟨*Sphere Method Definitions*⟩ +≡

```
Point Sphere::Sample(const Point &p, float u1, float u2,
                     Normal *ns) const {
    ⟨Compute coordinate system for sphere sampling 721⟩
    ⟨Sample uniformly on sphere if p is inside it 721⟩
    ⟨Sample sphere uniformly inside subtended cone 722⟩
}
```

This process is most easily done if we first compute a coordinate system to use for sampling the sphere where the z axis is the vector between the sphere's center and the point being illuminated:

Disk 129
Normal 65
Point 63
Sample 343
Shape::ObjectToWorld 108
Shape::
ReverseOrientation 108
Sphere::radius 116
UniformSampleSphere() 664
Vector::Normalize() 63

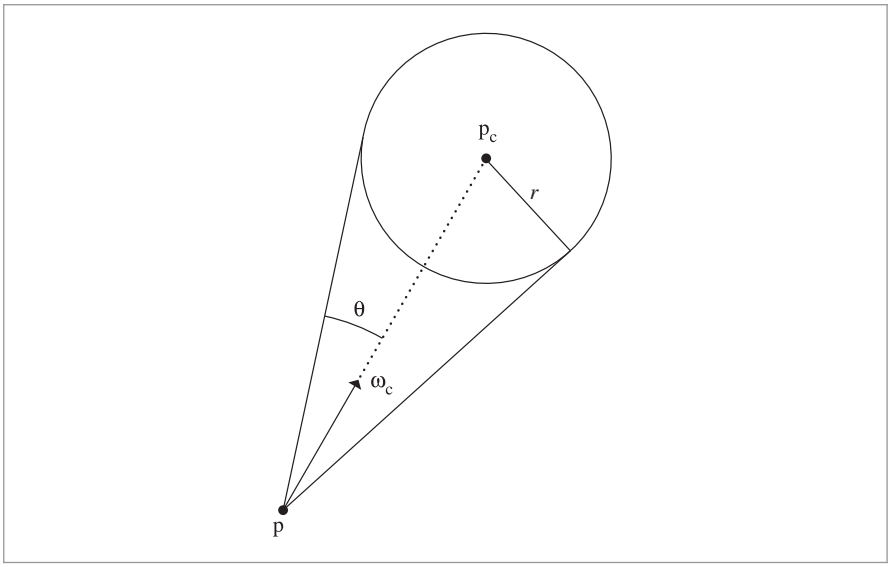


Figure 14.11: To sample points on a spherical light source, we can uniformly sample within the cone of directions around a central vector ω_c with an angular spread of up to θ. Trigonometry can be used to derive the value of $\sin\theta$, $r/|p_c - p|$.

⟨*Compute coordinate system for sphere sampling*⟩ ≡ 720
```
Point Pcenter = (*ObjectToWorld)(Point(0,0,0));
Vector wc = Normalize(Pcenter - p);
Vector wcX, wcY;
CoordinateSystem(wc, &wcX, &wcY);
```

It is important to be careful about points that lie inside the sphere. If this happens, the entire sphere should be sampled, since the whole sphere is clearly visible from inside it. Notice that we use a small constant, `1e-4`, to do this check; this makes the code more robust when points are very close to the sphere.

⟨*Sample uniformly on sphere if* p *is inside it*⟩ ≡ 720
```
if (DistanceSquared(p, Pcenter) - radius*radius < 1e-4f)
    return Sample(u1, u2, ns);
```

If the point being lit is outside the sphere, this method computes the cosine of the subtended angle $\theta_{\max}$ using Equation (14.2) and then generates a random ray inside the subtended cone using the `UniformSampleCone()` function and intersects it with the sphere to get the sample point on its surface.

Note that we must be careful about precision errors here. If the generated ray just grazes the edge of the sphere, the `Sphere::Intersect()` routine might unexpectedly return `false`. In this case, the implementation sets the `thit` value so that it corresponds to the point where the ray should have intersected the sphere at the edge.

CoordinateSystem() 63
DistanceSquared() 65
Point 63
Shape::ObjectToWorld 108
Sphere::Intersect() 117
Sphere::radius 116
Sphere::Sample() 720
UniformSampleCone() 713
Vector 57
Vector::Normalize() 63

⟨*Sample sphere uniformly inside subtended cone*⟩ ≡ 720
```
float sinThetaMax2 = radius*radius / DistanceSquared(p, Pcenter);
float cosThetaMax = sqrtf(max(0.f, 1.f - sinThetaMax2));
DifferentialGeometry dgSphere;
float thit, rayEpsilon;
Point ps;
Ray r(p, UniformSampleCone(u1, u2, cosThetaMax, wcX, wcY, wc), 1e-3f);
if (!Intersect(r, &thit, &rayEpsilon, &dgSphere))
    thit = Dot(Pcenter - p, Normalize(r.d));
ps = r(thit);
*ns = Normal(Normalize(ps - Pcenter));
if (ReverseOrientation) *ns *= -1.f;
return ps;
```

⟨*Sphere Method Definitions*⟩ +≡
```
float Sphere::Pdf(const Point &p, const Vector &wi) const {
    Point Pcenter = (*ObjectToWorld)(Point(0,0,0));
    ⟨Return uniform weight if point inside sphere 722⟩
    ⟨Compute general sphere weight 722⟩
}
```

To compute the PDF for sampling a direction towards a sphere from a given point, we must first determine which of the two sampling strategies for points inside and outside the sphere would be used for the point. If the shading point was inside the sphere, a uniform sampling strategy was used, in which case, the implementation hands off the `Pdf()` call to the parent class, which takes care of the solid angle conversion.

⟨*Return uniform weight if point inside sphere*⟩ ≡ 722
```
if (DistanceSquared(p, Pcenter) - radius*radius < 1e-4f)
    return Shape::Pdf(p, wi);
```

In the general case, we simply recompute the angle subtended by the sphere and call `UniformConePdf()`. Note that no conversion of sampling measures is required here because `UniformConePdf()` already returns weights with respect to a solid angle measure.

⟨*Compute general sphere weight*⟩ ≡ 722
```
float sinThetaMax2 = radius*radius / DistanceSquared(p, Pcenter);
float cosThetaMax = sqrtf(max(0.f, 1.f - sinThetaMax2));
return UniformConePdf(cosThetaMax);
```

14.6.4 SHAPESET SAMPLING

A `ShapeSet` is allocated in the `DiffuseAreaLight` constructor to store the shape or shapes that define the light. It conforms to the `Shape` interface so that the light source can just store a single `Shape` pointer, regardless of how many shapes are in the light source. The `ShapeSet` thus must implement the `Shape`'s sampling routines.

In order to be able to uniformly sample the overall shape by surface area, we would like to sample individual shapes in the collection with probability based on the ratio of their area to the total area of all of the shapes. Therefore, the `ShapeSet` constructor uses the

DifferentialGeometry 102
DiffuseAreaLight 625
DistanceSquared() 65
Normal 65
Point 63
Ray 66
Shape 108
Shape::ObjectToWorld 108
Shape::Pdf() 717
Shape::
ReverseOrientation 108
ShapeSet 626
Sphere::Intersect() 117
Sphere::radius 116
UniformConePdf() 713
UniformSampleCone() 713
Vector 57
Vector::Normalize() 63

Distribution1D to store a discrete CDF for sampling each shape according to individual probabilities given by this ratio.

⟨*Compute total area of shapes in* ShapeSet *and area CDF*⟩ ≡
```
sumArea = 0.f;
for (uint32_t i = 0; i < shapes.size(); ++i) {
    float a = shapes[i]->Area();
    areas.push_back(a);
    sumArea += a;
}
areaDistribution = new Distribution1D(&areas[0], areas.size());
```

⟨*ShapeSet Private Data*⟩ +≡ 626
```
float sumArea;
vector<float> areas;
Distribution1D *areaDistribution;
```

Here we'll show the variants of the ShapeSet's Sample() and Pdf() methods that take the point being shaded and sample by solid angle. The variants for the case of sampling with respect to area and sampling outgoing rays are similar and so are elided. For Sample(), a shape is selected according to its area and its Sample() method is called.

⟨*ShapeSet Method Definitions*⟩ ≡
```
Point ShapeSet::Sample(const Point &p, const LightSample &ls,
                       Normal *Ns) const {
    int sn = areaDistribution->SampleDiscrete(ls.uComponent, NULL);
    Point pt = shapes[sn]->Sample(p, ls.uPos[0], ls.uPos[1], Ns);
    ⟨Find closest intersection of ray with shapes in ShapeSet 723⟩
}
```

It may be that one of the other shapes is visible along the ray from the point being illuminated to the sampled point; in this case, it's important that we return the point on that shape rather than the originally sampled point. The implementation here checks for intersection with all of the shapes; this may be inefficient for large numbers of shapes.

⟨*Find closest intersection of ray with shapes in* ShapeSet⟩ ≡ 723
```
Ray r(p, pt-p, 1e-3f, INFINITY);
float rayEps, thit = 1.f;
bool anyHit = false;
DifferentialGeometry dg;
for (uint32_t i = 0; i < shapes.size(); ++i)
    anyHit |= shapes[i]->Intersect(r, &thit, &rayEps, &dg);
if (anyHit) *Ns = dg.nn;
return r(thit);
```

To compute the value of the PDF, we just need to iterate over all of the Shapes and call their Pdf() methods, taking the area-weighted average of their PDFs since shapes are selected by area in ShapeSet::Sample(). Executing this method will be inefficient when large numbers of lights are in the ShapeSet, due to the expense of evaluating the PDF for each of them (recall that a ray–shape intersection is performed in the Shape::Pdf()

DifferentialGeometry 102
DifferentialGeometry::nn 102
Distribution1D 648
Distribution1D::
SampleDiscrete() 650
INFINITY 1002
LightSample 710
LightSample::uComponent 710
LightSample::uPos 710
Normal 65
Point 63
Ray 66
Sample 343
Shape 108
Shape::Area() 113
Shape::Intersect() 111
Shape::Sample() 716
ShapeSet 626
ShapeSet::
areaDistribution 723
ShapeSet::areas 723
ShapeSet::Sample() 723
ShapeSet::shapes 626
ShapeSet::sumArea 723

methods). The ShapeSet could therefore maintain a small acceleration structure here so that it could avoid unnecessarily calling this method for the Shapes that the ray $(\mathrm{p}, \omega_\mathrm{i})$ doesn't intersect at all.

⟨*ShapeSet Method Definitions*⟩ +≡
```
float ShapeSet::Pdf(const Point &p, const Vector &wi) const {
    float pdf = 0.f;
    for (uint32_t i = 0; i < shapes.size(); ++i)
        pdf += areas[i] * shapes[i]->Pdf(p, wi);
    return pdf / sumArea;
}
```

14.6.5 INFINITE AREA LIGHTS

The InfiniteAreaLight, defined in Section 12.5, can be considered to be an infinitely large sphere that surrounds the entire scene, illuminating it from all directions. The environment maps used with InfiniteAreaLights often have substantial variation along different directions: consider, for example, an environment map of the sky during daytime, where the relatively small number of directions that the sun subtends will be thousands of times brighter than the rest of the directions. Given this substantial variation, implementing a sampling method for InfiniteAreaLights that matches the illumination distribution substantially reduces variance in images.

Figure 14.12 shows two images of the Audi TT car model illuminated by the morning skylight environment map from Figure 12.16. The top image was rendered using a simple cosine-weighted sampling distribution for selecting incident illumination directions, while the bottom image was rendered using the sampling method implemented in this section. Both images used just 32 shadow samples per pixel. For the same number of samples taken and with negligible additional computational cost, this sampling method computes a substantially better result with much lower variance.

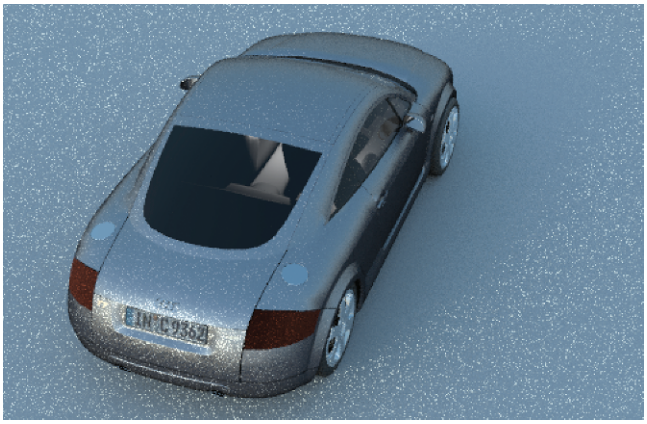
There are three main steps to the sampling approach implemented here:

1. We define a piecewise-constant 2D probability distribution function in (u, v) image coordinates $p(u, v)$ that corresponds to the distribution of the radiance represented by the environment map.
2. We apply the sampling method from Section 13.6.5 to transform uniformly distributed 2D random numbers to samples drawn from the piecewise-constant $p(u, v)$ distribution.
3. We define a probability density function over directions on the unit sphere based on the probability density over (u, v).

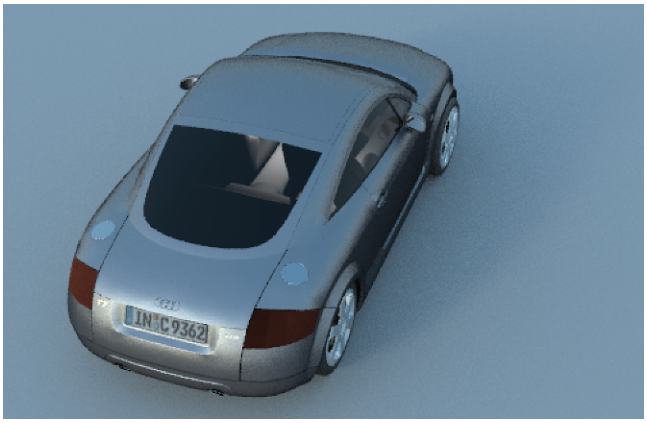
The combination of these three steps makes it possible to generate samples on the sphere of directions according to a distribution that matches the radiance function very closely, leading to substantial variance reduction.

We will start by defining the fragment ⟨*Initialize sampling PDFs for infinite area light*⟩ at the end of the InfiniteAreaLight constructor.

InfiniteAreaLight 629
Point 63
Shape 108
Shape::Pdf() 717
ShapeSet::areas 723
ShapeSet::shapes 626
ShapeSet::sumArea 723
Vector 57



(a)



(b)

Figure 14.12: TT Car Model Illuminated by the Morning Skylight Environment Map. These images were rendered with four image samples per pixel and eight light source samples per image sample. (a) The result of using a uniform sampling distribution, and (b) the improvement from the importance sampling method implemented here. A total of just 32 light samples per pixel gives an excellent result with this approach.

⟨*Initialize sampling PDFs for infinite area light*⟩ ≡ 630
```
    ⟨Compute scalar-valued image img from environment map 727⟩
    ⟨Compute sampling distributions for rows and columns of image 727⟩
    delete[] img;
```

Spectrum::y() 273

The first step is to transform the continuously defined spectral radiance function defined by the environment map's texels to a piecewise-constant scalar function by computing its luminance at a set of sample points using the `Spectrum::y()` method. There are three things to note in the code below that does this computation.

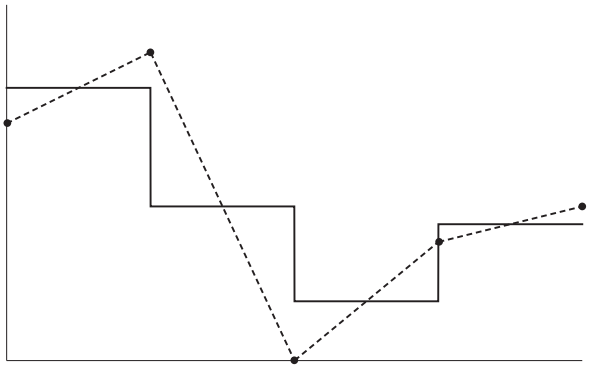


Figure 14.13: Finding a Piecewise-Constant Function (solid lines) that Approximates a Piecewise Linear Function (dashed lines) for Use as a Sampling Distribution. Even though some of the sample points that define the piecewise linear function (solid dots) may be zero-valued, the piecewise-constant sampling function must not be zero over any range where the actual function is nonzero. A reasonable approach to avoid this case, shown here and implemented in the `InfiniteAreaLight` sampling routines, is to find the average value of the function over some range and use that to define the piecewise-constant function.

First, it computes values of the radiance function at the same number of points as there are texels in the original image map. It could use either more or fewer points, leading to a corresponding increase or decrease in memory use while still generating a valid sampling distribution, however. These values work well, though, as fewer points would lead to a sampling distribution that didn't match the function as well while more would mostly waste memory with little incremental benefit.

The second thing of note in this code is that the piecewise constant function values being stored here in `img` are found by slightly blurring the radiance function with the `MIPMap::Lookup()` method (rather than just copying the corresponding texel values). The motivation for this is subtle; Figure 14.13 illustrates the idea in 1D. Because the continuous radiance function used for rendering is reconstructed by bilinearly interpolating between texels in the image, just because some texel is completely black, for example, the radiance function may be nonzero a tiny distance away from it due to a neighboring texel's contribution. Because we are using a piecewise-constant function for sampling rather than a piecewise-linear one, it must account for this issue in order to ensure greater-than-zero probability of sampling any point where the radiance function is nonzero.[3]

Finally, each image value in the `img` buffer is multiplied by the value of $\sin\theta$ corresponding to the θ value each row has when the latitude–longitude image is mapped to the sphere. Note that this multiplication has no effect on the sampling method's correctness:

3 Alternatively, we could use a piecewise-linear function for importance sampling and thus match the radiance function exactly. However, it's easier to draw samples from a piecewise-constant function's distribution, and, because environment maps generally have a large number of texel samples, a piecewise-constant function suffices to match its distribution well.

because the value of $\sin\theta$ is always greater than zero over the $[0, \pi]$ range, we are just reshaping the sampling PDF. The motivation for adjusting the PDF is to eliminate the effect of the distortion from mapping the 2D image to the unit sphere in the sampling method here; the details will be fully explained later in this section.

⟨*Compute scalar-valued image* `img` *from environment map*⟩ ≡ 725
```
float filter = 1.f / max(width, height);
float *img = new float[width*height];
for (int v = 0; v < height; ++v) {
    float vp = (float)v / (float)height;
    float sinTheta = sinf(M_PI * float(v+.5f)/float(height));
    for (int u = 0; u < width; ++u) {
        float up = (float)u / (float)width;
        img[u+v*width] = radianceMap->Lookup(up, vp, filter).y();
        img[u+v*width] *= sinTheta;
    }
}
```

Given this filtered and scaled image, the `Distribution2D` structure handles computing and storing the 2D PDF.

⟨*Compute sampling distributions for rows and columns of image*⟩ ≡ 725
```
distribution = new Distribution2D(img, width, height);
```

⟨*InfiniteAreaLight Private Data*⟩ +≡ 629
```
Distribution2D *distribution;
```

Given this precomputed data, the task of the sampling method is relatively straightforward. Given a pair of uniformly distributed random variables (ξ_1, ξ_2) over $[0, 1]^2$, it draws a sample from the function's distribution using the sampling algorithm described in Section 13.6.5, giving a (u, v) value and the value of the PDF for taking this sample, $p(u, v)$.

⟨*InfiniteAreaLight Method Definitions*⟩ +≡
```
Spectrum InfiniteAreaLight::Sample_L(const Point &p, float pEpsilon,
        const LightSample &ls, float time, Vector *wi, float *pdf,
        VisibilityTester *visibility) const {
    ⟨Find (u, v) sample coordinates in infinite light texture 727⟩
    ⟨Convert infinite light sample point to direction 728⟩
    ⟨Compute PDF for sampled infinite light direction 728⟩
    ⟨Return radiance value for infinite light direction 729⟩
}
```

⟨*Find* (u, v) *sample coordinates in infinite light texture*⟩ ≡ 727
```
float uv[2], mapPdf;
distribution->SampleContinuous(ls.uPos[0], ls.uPos[1], uv, &mapPdf);
```

The (u, v) sample is mapped to spherical coordinates by

$$(\theta, \phi) = (\pi v, 2\pi u) ,$$

and then the spherical coordinates formula gives the direction $\omega = (x, y, z)$.

Distribution2D 672
Distribution2D::SampleContinuous() 673
InfiniteAreaLight::distribution 727
LightSample 710
LightSample::uPos 710
MIPMap::Lookup() 540
M_PI 1002
Point 63
Spectrum 263
Spectrum::y() 273
Vector 57
VisibilityTester 608

⟨*Convert infinite light sample point to direction*⟩ ≡ 727

```
float theta = uv[1] * M_PI, phi = uv[0] * 2.f * M_PI;
float costheta = cosf(theta), sintheta = sinf(theta);
float sinphi = sinf(phi), cosphi = cosf(phi);
*wi = LightToWorld(Vector(sintheta * cosphi, sintheta * sinphi,
                          costheta));
```

Recall that the probability density values returned by the light source sampling routines must be defined in terms of the solid angle measure on the unit sphere. Therefore, this routine must now compute the transformation between the sampling density used, which was the image function over $[0, 1]^2$, and the corresponding density after the image has been mapped to the unit sphere with the latitude–longitude mapping. (Recall that the latitude–longitude parameterization of an image (θ, ϕ) is $x = r \sin\theta \cos\phi$, $y = r \sin\theta \sin\phi$, and $z = r \cos\theta$.)

First, consider the function g that maps from (u, v) to (θ, ϕ),

$$g(u, v) = (\pi v, 2\pi u) .$$

The absolute value of the determinant of the Jacobian $|J_g|$ is $2\pi^2$. Applying the multidimensional change of variables equation from Section 13.5.1, we can find the density in terms of spherical coordinates (θ, ϕ).

$$p(\theta, \phi) = \frac{p(u, v)}{2\pi^2}.$$

From the definition of spherical coordinates, it is easy to determine that the absolute value of the Jacobian for the mapping from (r, θ, ϕ) to (x, y, z) is $r^2 \sin\theta$. Since we are interested in the unit sphere, $r = 1$, and again applying the multidimensional change of variables equation to find the final relationship between probability densities,

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

This is the key relationship for applying this technique: it lets us sample from the piecewise-constant distribution defined by the image map and transform the sample and its probability density to be in terms of directions on the unit sphere.

We can now see why the initialization routines multiplied the values of the piecewise-constant sampling function by a $\sin\theta$ term. Consider, for example, a constant-valued environment map: with the $p(u, v)$ sampling technique, all (θ, ϕ) values are equally likely to be chosen. Due to the mapping to directions on the sphere, however, this would lead to more directions being sampled near the poles of the sphere, *not* a uniform sampling of directions on the sphere, which would be a more desirable result. The $1/\sin\theta$ term in the $p(\omega)$ PDF corrects for this non-uniform sampling of directions so that correct results are computed in Monte Carlo estimates. Given this state of affairs, however, it's better to have modified the $p(u, v)$ sampling distribution so that it's less likely to select directions near the poles in the first place.

Light::LightToWorld 606
M_PI 1002
Vector 57

⟨*Compute PDF for sampled infinite light direction*⟩ ≡ 727

```
*pdf = mapPdf / (2.f * M_PI * M_PI * sintheta);
if (sintheta == 0.f) *pdf = 0.f;
```

We can finally initialize the VisibilityTester and return the radiance value for the chosen direction.

⟨*Return radiance value for infinite light direction*⟩ ≡ 727
```
visibility->SetRay(p, pEpsilon, *wi, time);
Spectrum Ls = Spectrum(radianceMap->Lookup(uv[0], uv[1]),
                       SPECTRUM_ILLUMINANT);
return Ls;
```

The InfiniteAreaLight::Pdf() method needs to convert the direction ω to the corresponding (u, v) coordinates in the sampling distribution. Given these, the PDF $p(u, v)$ is computed as the product of the two 1D PDFs by the Distribution2D::Pdf() method, which is adjusted here for mapping to the sphere as was done in the Sample_L() method.

⟨*InfiniteAreaLight Method Definitions*⟩ +≡
```
float InfiniteAreaLight::Pdf(const Point &, const Vector &w) const {
    Vector wi = WorldToLight(w);
    float theta = SphericalTheta(wi), phi = SphericalPhi(wi);
    float sintheta = sinf(theta);
    if (sintheta == 0.f) return 0.f;
    float p = distribution->Pdf(phi * INV_TWOPI, theta * INV_PI) /
           (2.f * M_PI * M_PI * sintheta);
    return p;
}
```

Generating a random ray leaving an infinite light source can be done by sampling a direction with the same approach as is used for direct lighting in the Sample_L() method above. The sampled ray's origin is then set using the same approach as was used for DistantLights in Section 14.6.2, where a disk that covers the scene's bounding sphere is oriented along the ray's direction. (Recall Figure 14.9.) We therefore won't include the ⟨*Compute direction for infinite light sample ray*⟩ or ⟨*Compute origin for infinite light sample ray*⟩ fragments here.

⟨*InfiniteAreaLight Method Definitions*⟩ +≡
```
Spectrum InfiniteAreaLight::Sample_L(const Scene *scene,
        const LightSample &ls, float u1, float u2, float time,
        Ray *ray, Normal *Ns, float *pdf) const {
    ⟨Compute direction for infinite light sample ray⟩
    ⟨Compute origin for infinite light sample ray⟩
    ⟨Compute InfiniteAreaLight ray PDF 730⟩
    Spectrum Ls = (radianceMap->Lookup(uv[0], uv[1]), SPECTRUM_ILLUMINANT);
    return Ls;
}
```

The PDF for these rays is just the product of the PDF for sampling the direction and the PDF for sampling a point on the disk.

DistantLight 621
Distribution2D::Pdf() 673
InfiniteAreaLight 629
InfiniteAreaLight::Pdf() 729
INV_PI 1002
INV_TWOPI 1002
Light::WorldToLight 606
LightSample 710
MIPMap::Lookup() 540
M_PI 1002
Normal 65
Point 63
Ray 66
Scene 22
Spectrum 263
SPECTRUM_ILLUMINANT 277
SphericalPhi() 292
SphericalTheta() 292
Vector 57
VisibilityTester 608
VisibilityTester::
  SetRay() 609

⟨*Compute* InfiniteAreaLight *ray PDF*⟩ ≡ 729

```
float directionPdf = mapPdf / (2.f * M_PI * M_PI * sintheta);
float areaPdf = 1.f / (M_PI * worldRadius * worldRadius);
*pdf = directionPdf * areaPdf;
if (sintheta == 0.f) *pdf = 0.f;
```

* 14.7 VOLUME SCATTERING

The equation of transfer is the integral equation that describes the effect of participating media on the distribution of radiance in an environment. It is introduced in Section 16.1, and the VolumeIntegrators in Chapter 16 use Monte Carlo integration to compute estimates of (simplified) instances of this integral. The equation of transfer is an interesting example case for Monte Carlo integration since it can be an integral over an infinite range, of the general form

$$\int_0^\infty \mathrm{e}^{-g(t)t} f(t)\, \mathrm{d}t.$$

Thanks to the exponential term, this integral has a finite value in volume scattering applications.

It's not possible to sample from a uniform distribution to compute estimates of this integral. There's no valid uniform PDF over the range $[0, \infty)$, so some other sampling distribution must be used instead. In some cases, the function $g(t)$ is known to be constant and a natural sampling distribution is the one based on the exponential term; Section 13.3.1 derived the general sampling process. Thus, if samples T_i are drawn from this distribution, estimates of the integral can be computed by

$$\frac{1}{N}\sum_i^N \frac{\mathrm{e}^{-cT_i} f(T_i)}{c\,\mathrm{e}^{-cT_i}} = \frac{1}{N}\sum_i^N \frac{f(T_i)}{c}.$$

More generally, if the function $g(t)$ is not constant, a value of c for the exponential sampling distribution can be chosen arbitrarily (e.g., based on the average value of $g(t)$, if known), and the estimate is

$$\frac{1}{N}\sum_i^N \frac{\mathrm{e}^{-g(T_i)T_i} f(T_i)}{c\,\mathrm{e}^{-cT_i}}.$$

Because the extents of VolumeRegions in pbrt must be finite, the system never actually needs to estimate values of these integrals over infinite ranges, and a uniform sampling distribution can still be used. However, as the extents of VolumeRegions increase and as the $g(t)$ term becomes larger, uniform sampling is an increasingly inefficient sampling method as samples are taken in areas where the exponential function is very small, and the variance from not using importance sampling based on the exponential term increases. The implementations of VolumeIntegrators in Chapter 16 use uniform sampling, although Exercises 16.3, 16.4, and 16.5 at the end of that chapter discuss how to address this issue.

M_PI 1002
VolumeIntegrator 876
VolumeRegion 587

14.7.1 SAMPLING PHASE FUNCTIONS

It is useful to be able to draw samples from the distribution described by the Henyey–Greenstein phase functions for a number of applications, including applying importance sampling to computing direct lighting in participating media as well as for more general algorithms that compute the effect of multiple scattering in participating media. The PDF for this distribution is separable into θ and ϕ components, with $p(\phi) = 1/(2\pi)$ as usual. The distribution for θ can be shown to be

$$\cos\theta = \frac{1}{2g}\left(1 + g^2 - \left(\frac{1-g^2}{1-g+2g\xi}\right)^2\right)$$

if $g \neq 0$; otherwise, $\cos\theta = 1 - 2\xi$.

This sampling technique is implemented here in a routine that takes one vector ω, the asymmetry parameter g, and two random numbers; samples from this distribution; and uses the result to construct the corresponding outgoing ray.

⟨*Monte Carlo Function Definitions*⟩ +≡
```
Vector SampleHG(const Vector &w, float g, float u1, float u2) {
    float costheta;
    if (fabsf(g) < 1e-3)
        costheta = 1.f - 2.f * u1;
    else {
        float sqrTerm = (1.f - g * g) /
                        (1.f - g + 2.f * g * u1);
        costheta = (1.f + g * g - sqrTerm * sqrTerm) / (2.f * g);
    }
    float sintheta = sqrtf(max(0.f, 1.f-costheta*costheta));
    float phi = 2.f * M_PI * u2;
    Vector v1, v2;
    CoordinateSystem(w, &v1, &v2);
    return SphericalDirection(sintheta, costheta, phi, v1, v2, w);
}
```

Because phase functions are already distribution functions, the probability density for sampling any particular direction is the value of the phase function for the given pair of directions.

⟨*Monte Carlo Function Definitions*⟩ +≡
```
float HGPdf(const Vector &w, const Vector &wp, float g) {
    return PhaseHG(w, wp, g);
}
```

CoordinateSystem() 63
M_PI 1002
PhaseHG() 585
SphericalDirection() 292
Vector 57

14.7.2 COMPUTING OPTICAL THICKNESS

Section 11.1.3 introduced optical thickness τ, which gives a measure of the total density of participating media that a ray passes through. For media with constant scattering

properties, optical thickness can be computed in closed form using Beer's law, Equation (11.1), although that chapter deferred the implementation of a method to compute τ for general media until after Monte Carlo integration had been introduced. Here, we will implement the general `DensityRegion::tau()` method, which computes an estimate of the optical thickness for a ray passing through an arbitrary medium.

Recall the definition of τ:

$$\tau(\mathrm{p} \to \mathrm{p}') = \int_0^d \sigma_\mathrm{t}(\mathrm{p} + t\omega, -\omega)\, \mathrm{d}t.$$

Its value can clearly be estimated by

$$\frac{1}{N} \sum_i^N \frac{\sigma_\mathrm{t}(\mathrm{p} + T_i\, \omega,\ -\omega)}{p(T_i)},$$

where random variables T_i are sampled from some distribution p. A natural approach is to apply stratified sampling, dividing the line from 0 to d into N strata and placing one random sample in each one. This could be done by transforming uniform random samples ξ by $d\xi$.

However, recall that Monte Carlo's strength compared to standard numerical integration algorithms comes when we are faced with high-dimensional integrals and discontinuous integrands. Here, we have a one-dimensional integral and an integrand that is often smoothly varying. (Consider the spatially varying density of a cloud or smoke, for example.) Pauly and collaborators have shown that a more efficient integration technique for this problem is to generate a stratified pattern where the sample inside each stratum has the same offset (Pauly 1999; Pauly, Kollig, and Keller 2000):

$$T_i = \frac{\xi + i}{N} d.$$

A single uniform random variable ξ is used for all of the samples. The result is that samples in adjacent strata can't be bunched together, as can happen with regular stratified sampling, thus leading to reduced variance.

The implementation of the `tau()` method here applies this technique, which is why it only takes a single random variable `u`. It also indirectly chooses the number of samples N to take for the estimator. The caller passes in `stepSize`, the desired size of the strata, and the method finds the length of the given ray overlapping the medium, d, and then determines N by

$$N = \left\lceil \frac{d}{\mathrm{stepSize}} \right\rceil.$$

N is not computed explicitly in the following implementation. Instead, the `while` loop will execute once for each sample taken until the point `t0` exits the region.

Mathematically, the overall estimate should be divided by N and the (constant) PDF value. However, the PDF is just $1/d = 1/(t_1 - t_0)$, so the division by Nd can be simply computed by multiplying by the step size at the end.

DensityRegion::tau() 733

⟨*Volume Scattering Definitions*⟩ +≡

```
Spectrum DensityRegion::tau(const Ray &r, float stepSize,
                            float u) const {
    float t0, t1;
    float length = r.d.Length();
    if (length == 0.f) return 0.f;
    Ray rn(r.o, r.d / length, r.mint * length, r.maxt * length, r.time);
    if (!IntersectP(rn, &t0, &t1)) return 0.;
    Spectrum tau(0.);
    t0 += u * stepSize;
    while (t0 < t1) {
        tau += sigma_t(rn(t0), -rn.d, r.time);
        t0 += stepSize;
    }
    return tau * stepSize;
}
```

FURTHER READING

Cook and collaborators first introduced random sampling for integration in rendering (Cook, Porter, and Carpenter 1984; Cook 1986), and Kajiya (1986) developed the general-purpose path-tracing algorithm. Other important early work on Monte Carlo in rendering includes Shirley's Ph.D. thesis (Shirley 1990) and articles by Arvo and Kirk on Russian roulette, splitting, and sources of bias in rendering algorithms (Kirk and Arvo 1991). The multiple importance sampling technique was invented by Veach and Guibas (Veach and Guibas 1995; Veach 1997).

Shirley and collaborators have written a number of key papers about Monte Carlo in graphics, including a survey of techniques for light source sampling (Shirley, Wang, and Zimmerman 1996) and recipes for warping uniform random numbers to the surfaces of various shapes (Shirley 1992). See also the "Further Reading" section in Chapter 15 for additional discussion of Monte Carlo techniques for direct lighting.

Mitchell (1996b) wrote a paper that investigates the effects of stratification for integration problems in graphics (including the two-dimensional problem of pixel antialiasing). In particular, he investigated the connection between the complexity of the function being integrated and the effect of stratification. In general, the smoother or simpler the function, the more stratification helps: for pixels with smooth variation over their areas or with just a few edges passing through them, stratification helps substantially, but as the complexity in a pixel is increased the gain from stratification is reduced. Nevertheless, because stratification never increases variance, there's no reason not to do it.

Keller and collaborators have written extensively on the application of quasi–Monte Carlo integration in graphics (Keller 1996, 2001; Friedel and Keller 2000; Kollig and Keller 2000, 2002), and Owen and Tribble (2005) discussed the requirements for correctly using quasi-random patterns with Metropolis sampling.

DensityRegion 591
Ray 66
Spectrum 263
Vector::Length() 62
VolumeRegion::
IntersectP() 588
VolumeRegion::sigma_t() 588

Lawrence et al. (2004) developed methods for sampling arbitrary BRDF models, including those based on measured reflectance data. They apply methods that factor the 4D BRDF into a product of two 2D functions, both of which are guaranteed to always be greater than zero, thus making it possible to use them as importance sampling distributions.

Subr and Arvo (2007) developed an efficient technique for sampling environment map light sources that accounts for both the $\cos\theta$ term from the scattering equation and also only generates samples in the hemisphere around the surface normal. A number of papers have been written about techniques transforming environment lights (like `InfiniteAreaLight`) into collections of distant light sources. See, for example, the papers by Agarwal et al. (2003) and Kollig and Keller (2003). This approach introduces additional correlation (using the same light sample positions at all shaded points), though the potential artifacts (banded shadows and reflections) may be less objectionable than Monte Carlo noise.

Talbot et al. (2005) applied *importance resampling* to rendering, showing that this variant of standard importance sampling is applicable to a number of problems in graphics, and Pegoraro et al. (2008a) implemented an approach that found sampling PDFs for global illumination over the course of rendering the image.

A number of researchers have recently developed efficient methods to directly sample the distribution given by the product of two functions (for example, the BSDF and an infinite area light source). See the "Further Reading" section in Chapter 15 for discussion of this work.

EXERCISES

❷ 14.1 One shortcoming of the current implementation of the `BSDF::Sample_f()` method is that if some of the `BxDF`s make a much larger contribution to the overall result than others, then uniformly choosing among them to determine a sampling distribution may be suboptimal. Modify this method so that it instead chooses among the `BxDF`s according to their relative reflectances. (Don't forget to also update `BSDF::Pdf()` to account for this change.) Can you create a contrived set of parameters to a `Material` that causes this approach to be substantially better than the built-in one? Does this change have a noticeable effect on Monte Carlo efficiency for typical scenes?

❸ 14.2 Implement the sampling method for arbitrary BRDFs introduced by Lawrence et al. (2005) as the sampling approach used for one or both of the measured BRDF implementations in Section 8.6. Perform tests using different sets of BRDF data and measure the improvement in Monte Carlo efficiency (variance divided by computation time) for different data sets.

❶ 14.3 Fix the buggy `Sphere::Sample()` and `Disk::Sample()` methods, which currently don't properly account for partial spheres and disks when they sample points on the surface. Create a scene that demonstrates the error from the current implementations and for which your solution is clearly an improvement.

BSDF::Pdf() 708
BSDF::Sample_f() 706
BxDF 428
Disk::Sample() 719
InfiniteAreaLight 629
Material 483
Sphere::Sample() 720

2 **14.4** It is possible to derive a sampling method for cylinder area light sources that only chooses points over the visible area as seen from the receiving point, similar to the improved sphere sampling method in this chapter (Gardner et al. 1987; Zimmerman 1995). Learn more about these methods, or rederive them yourself, and write a new implementation of `Cylinder::Sample()` that implements such an algorithm. Verify that `pbrt` still generates correct images with your method and measure how much the improved version reduces variance for a fixed number of samples taken. How much does it improve efficiency? How do you explain any discrepancy between the amount of reduction in variance and the amount of improvement in efficiency?

2 **14.5** Discuss situations where the current methods for sampling outgoing rays from `ProjectionLights` and `GonioPhotometricLights` may be extremely inefficient, choosing many rays in directions where the light source casts no illumination. Use the `Distribution2D` structure to implement improved sampling techniques for each of them based on sampling from a distribution based on the luminance in their two-dimensional image maps, properly accounting for the transformation from the 2D image map sampling distribution to the distribution of directions on the sphere. Verify that the system still computes the same images (modulo variance) with your new sampling techniques when using an `Integrator` that calls these methods. Determine how much efficiency is improved by using these sampling methods instead of the default ones.

3 **14.6** Read the papers by Agarwal et al. (2003) or Kollig and Keller (2003) and implement one of these approaches to environment light sampling. Compare performance of the approach implemented in `pbrt` to this new approach. Discuss the trade-offs among them.

3 **14.7** One useful technique not discussed in this chapter is the idea of adaptive density distribution functions that dynamically change the sampling distribution as samples are taken and information is available about the integrand's actual distribution as a result of evaluating the values of these samples. The standard Monte Carlo estimator can be written to work with a nonuniform distribution of random numbers used in a transformation method to generate samples X_i,

$$\sum_i^N \frac{f(X_i)}{p(X_i)p_r(\xi_i)},$$

just like the transformation from one sampling density to another. This leads to a useful sampling technique, where an algorithm can track which samples ξ_i were effective at finding large values of $f(x)$ and which weren't and then adjusts probabilities toward the effective ones (Booth 1986). A straightforward implementation would be to split $[0, 1]$ into bins of fixed width, track the average value of the integrand in each bin, and use this to change the distribution of ξ_i samples.

Investigate data structures and algorithms that support such sampling approaches and choose a sampling problem in `pbrt` to apply them to. Measure how well this approach works for the problem you selected. One difficulty with

Cylinder::Sample() 719
Distribution2D 672
GonioPhotometricLight 618
Integrator 740
ProjectionLight 614

methods like this is that different parts of the sampling domain will be the most effective at different times in different parts of the scene. For example, trying to adaptively change the sampling density of points over the surface of an area light source has to contend with the fact that, at different parts of the scene, different parts of the area light may be visible and thus be the important areas. You may find it useful to read Cline et al.'s paper (2008) on this topic.

2 14.8 `pbrt`'s routines for using complex shapes like triangle meshes or subdivision meshes as area light sources are written to handle simple shapes and do not scale well to shapes that refine into more than a few tens of triangles in the end. An effective way to sample shapes such as these is to uniformly sample rays from the cone that surrounds the shape as seen from the point being shaded, see if the ray hits the shape, and use the hit location, if any, as the sample point on the area source.

Implement such a sampling method for complex shapes. Test your implementation by using it to render images with simple shapes and then render images using complex shapes as area light sources.

2 14.9 If additional high-level information is provided about the characteristics of the scene, there are various ways that rendering can be done more efficiently. For example, consider an infinite area light representing the sky illuminating a room through windows. If the scene description also includes shapes that represent the window openings and if we know that all light paths from the sky to the interior of the room light go through a window, then another way to sample directions to the light source is to sample by area on the surfaces of the window shapes (represented as a collection of triangles, for example) and then use the corresponding directions to the light, converting from the measure over area to one over directions on the unit sphere, similarly to what is done for area lights attached to `Shapes`. This approach can save taking many samples and tracing the corresponding rays that would otherwise certainly be occluded by walls or other geometry.

Implement this approach as an optional `InfiniteAreaLight` sampling method in `pbrt` and measure the improvement in efficiency for rendering scenes where there is a substantial amount of occlusion between the part of the scene being rendered and the infinite light source. (Results can be improved by using this as yet another light sampling technique alongside the regular light source sampling and BSDF sampling used in Section 15.1.1 and then weighted using multiple importance sampling.)

2 14.10 The infinite area light importance sampling method implemented in this chapter doesn't perfectly match the distribution of the light source's emission distribution: recall that the emission function is computed with bilinear interpolation among image samples but the sampling distribution is computed as a piecewise-constant function of a slightly blurred version of the texture map. In some cases, this discrepancy can lead to high variance when there are localized extremely bright texels. (In the worst case, consider a source that has a very small value at all texels except one, which is 10,000 times brighter than all of the

`InfiniteAreaLight` 629
`Shape` 108

others.) In that case, the value of the function may be much higher than predicted by the sampling PDFs at points very close to that sample, thus leading to high variance ($f(x)/p(x)$ is large).

Construct an environment map where this problem manifests itself in `pbrt` and fix the system so that the excessive variance goes away. One option is to modify the system so that the sampling distribution and the illumination function match perfectly by point sampling the environment map for lookups, rather than using bilinear filtering, though this can lead to undesirable image artifacts in the environment map, especially when directly visible from the camera. Can you find a way to only use the point-sampled environment map for direct lighting calculations but to use bilinear filtering for camera rays and rays that have only undergone specular reflection?

The other alternative is to modify the sampling distribution so that it matches the bilinearly filtered environment map values perfectly. Exact 2D sampling distributions for bilinear functions can be computed, though finding the sample value corresponding to a random variable is more computationally expensive, requiring solving a quadratic equation. Can you construct a scene where this overhead is worthwhile in return for the resulting variance reduction?