CHAPTER FIFTEEN

15 LIGHT TRANSPORT I: SURFACE REFLECTION

This chapter brings together the ray-tracing algorithms, radiometric concepts, and Monte Carlo sampling algorithms of the previous chapters to implement a set of integrators that compute scattered radiance from surfaces in the scene. Integrators are so named because they are responsible for evaluating the integral equation that describes the equilibrium distribution of radiance in an environment (the light transport equation). As the `SamplerRenderer` uses the `Camera` to generate rays and then finds intersections with scene geometry, information about the intersections is passed to the `SurfaceIntegrator` and the `VolumeIntegrator` that the user selected; together these two classes are responsible for doing appropriate shading and lighting computations to compute the radiance along the ray, accounting for light reflected from the first surface visible along the ray as well as light attenuated and scattered by participating media along the ray. Surface integrators are covered in this chapter, and volume integrators are the topic of Chapter 16. This chapter concludes with the implementation of a `Renderer` that applies the Metropolis sampling approach introduced in Section 13.4 to the light transport problem.

Because the light transport equation can be solved in closed form only for trivial scenes, it's necessary to apply a numerical integration technique to approximate its solution. How best to do so has been an active area of research in rendering, and many solution methods have been proposed. In this chapter, we will present implementations of a number of different integrators based on Monte Carlo integration that represent a selection of representative approaches to the problem. Due to basic decisions made in `pbrt`'s design, this chapter does not include methods based on finite-element algorithms (e.g., radiosity), which is the other major approach to solving the light transport equation. See the "Further Reading" section for more information about this method.

The basic integrator interfaces are defined in core/integrator.h, and some utility functions used by integrators are in core/integrator.cpp. The implementations of the various integrators are in the integrators/ directory.

Both surface and volume integrators inherit from the Integrator abstract base class.

⟨*Integrator Declarations*⟩ ≡
```
class Integrator {
public:
    ⟨Integrator Interface 740⟩
};
```

The integrator implementation may optionally implement the Preprocess() method. It is called after the Scene has been fully initialized and gives the integrator a chance to do scene-dependent computation, such as allocating additional data structures that are dependent on the number of lights in the scene, or precomputing a rough representation of the distribution of radiance in the scene. Integrators that don't need to do anything along these lines can leave this method unimplemented.

⟨*Integrator Interface*⟩ ≡ 740
```
virtual void Preprocess(const Scene *scene, const Camera *camera,
                        const Renderer *renderer) {
}
```

If the integrator would like the Sampler to generate sample patterns in the Sample for it to use, it should implement the RequestSamples() method and call back to the Sample::Add1D() and Sample::Add2D() methods, as described in Section 7.2.1.

⟨*Integrator Interface*⟩ +≡ 740
```
virtual void RequestSamples(Sampler *sampler, Sample *sample,
                            const Scene *scene) {
}
```

The SurfaceIntegrator interface class adds a single required method beyond those in the Integrator.

⟨*Integrator Declarations*⟩ +≡
```
class SurfaceIntegrator : public Integrator {
public:
    ⟨SurfaceIntegrator Interface 741⟩
};
```

The key method that all surface integrators must implement is SurfaceIntegrator::Li(), which returns the outgoing radiance at the intersection point of a given ray with the scene geometry, or, equivalently, incident radiance at the origin of the ray. (Recall that radiance is unchanged as it passes through a vacuum; VolumeIntegrators will separately compute the effect of attenuation through participating media between the ray origin and the intersection point.)

The parameters to this method are the following:

- scene—A pointer to the Scene being rendered. The integrator will query the scene for information about the lights and geometry, and so on.

Camera 302
Integrator 740
Renderer 24
Sample 343
Sample::Add1D() 344
Sample::Add2D() 344
Sampler 340
Scene 22
SurfaceIntegrator 740
SurfaceIntegrator::Li() 741
VolumeIntegrator 876

- `renderer`—A pointer to the `Renderer` being used for rendering. The integrator may call the `Li()` or `Transmittance()` methods of the renderer, which in turn compute radiance and transmittance along arbitrary rays including both surface and volume scattering effects.[1]
- `ray`—The ray along which the incident radiance should be evaluated.
- `isect`—An `Intersection` object encapsulating information about the first intersection of the ray with the scene geometry.
- `sample`—A pointer to a `Sample` generated by the `Sampler` for this ray. Some integrators will use information in this structure for Monte Carlo sampling.
- `arena`—A `MemoryArena` for efficient temporary memory allocation by the integrator. The integrator should assume that any memory it allocates with the arena will be freed shortly after the `Li()` method returns and thus should not use the arena to allocate any memory that must persist for longer than is needed for the current ray.

The method returns a `Spectrum` that represents the radiance along the ray:

⟨*SurfaceIntegrator Interface*⟩ ≡ 740
```
virtual Spectrum Li(const Scene *scene, const Renderer *renderer,
    const RayDifferential &ray, const Intersection &isect,
    const Sample *sample, RNG &rng, MemoryArena &arena) const = 0;
```

15.1 DIRECT LIGHTING

Before we introduce the light transport equation in its full generality, we will implement an integrator that accounts for only direct lighting—light that has traveled directly from a light source to the point being shaded—and ignores indirect illumination from objects that are not themselves emissive. Starting out with this integrator allows us to focus on some of the key details of direct lighting without worrying about the full light transport equation. Furthermore, some of the routines developed here will be used again in subsequent integrators that solve the complete light transport equation. Figure 15.1 shows the San Miguel scene rendered with direct lighting only.

The implementation here provides two different strategies for computing direct lighting. Each method computes an unbiased estimate of exitant radiance at a point in a given direction. The `LightStrategy` enumerant records which approach has been selected. The first strategy loops over all of the lights and takes a number of samples based on `Light::nSamples` from each of them, summing the result. The second takes a single sample from just one of the lights, chosen at random.

Depending on the scene being rendered, either of these approaches may be more appropriate. For example, if many image samples are being taken for each pixel (e.g., due to depth of field), a single light sample may be more appropriate, since in the aggregate they will sample the direct lighting enough to give a high-quality image. Alternatively, if few image samples are being taken, sampling all lights may be preferable.

CreateRadianceProbes 958
Intersection 186
Light::nSamples 606
MemoryArena 1015
RayDifferential 69
Renderer 24
RNG 1003
Sample 343
Sampler 340
SamplerRenderer 25
Scene 22
Spectrum 263

1 The `Renderer` will usually be a `SamplerRenderer` instance, but other `Renderer` implementations like `CreateRadianceProbes` that use instances of `Integrators` from this chapter will be the `Renderer` in that case.

Figure 15.1: Scene Rendered with Direct Lighting Only. Because only direct lighting is considered, some portions of the image are completely black because they are only lit by indirect illumination. *(Model courtesy of Guillermo M. Leal Llaguno.)*

⟨*DirectLightingIntegrator Declarations*⟩ ≡
```
enum LightStrategy { SAMPLE_ALL_UNIFORM, SAMPLE_ONE_UNIFORM };
```

⟨*DirectLightingIntegrator Declarations*⟩ +≡
```
class DirectLightingIntegrator : public SurfaceIntegrator {
public:
    ⟨DirectLightingIntegrator Public Methods⟩
private:
    ⟨DirectLightingIntegrator Private Data 742⟩
};
```

The implementation of the constructor isn't included here. It initializes the lighting strategy with a value passed in.

⟨*DirectLightingIntegrator Private Data*⟩ ≡ 742
```
LightStrategy strategy;
```

The member variables that record the sample information for direct lighting are declared with a level of indirection in a separate fragment, so that other integrators later in this chapter that use the direct lighting functions defined here can easily incorporate the appropriate member variables.

⟨*DirectLightingIntegrator Private Data*⟩ +≡ 742 SurfaceIntegrator 740
```
    ⟨Declare sample parameters for light source sampling 743⟩
int lightNumOffset;
```

The number and types of samples needed by this integrator depend on the sampling strategy used: if a single sample is taken from a single light, then only one of each of the `LightSampleOffsets` and `BSDFSampleOffsets` object needs to be created; otherwise, one is allocated for each light source in the scene. These structures handle the mechanics of informing the sampler of how many samples and of what dimensionality are needed—they are defined in Sections 14.6.1 and 14.5.6, respectively. Sample patterns are needed both to select points on light sources and to select BSDF directions for the direct lighting computation; both sampling approaches are combined with multiple importance sampling in the implementation below.

⟨*Declare sample parameters for light source sampling*⟩ ≡ 742, 788, 803
```
LightSampleOffsets *lightSampleOffsets;
BSDFSampleOffsets *bsdfSampleOffsets;
```

⟨*DirectLightingIntegrator Method Definitions*⟩ ≡
```
void DirectLightingIntegrator::RequestSamples(Sampler *sampler,
        Sample *sample, const Scene *scene) {
    if (strategy == SAMPLE_ALL_UNIFORM) {
        ⟨Allocate and request samples for sampling all lights 743⟩
        lightNumOffset = -1;
    }
    else {
        ⟨Allocate and request samples for sampling one light 744⟩
    }
}
```

If all of the lights are being sampled, the integrator needs `nLights` individual sampling patterns from the `Sampler`, each one sized according to the number of samples in that light's `nSamples` member variable. Note that the integrator only uses that value as a starting point. The `Sampler::RoundSize()` method is given an opportunity to change that value to a more appropriate one based on its particular sample generation technique. (For example, some samplers only generate collections of samples with power-of-two sizes.)

⟨*Allocate and request samples for sampling all lights*⟩ ≡ 743, 774, 788, 803
```
uint32_t nLights = scene->lights.size();
lightSampleOffsets = new LightSampleOffsets[nLights];
bsdfSampleOffsets = new BSDFSampleOffsets[nLights];
for (uint32_t i = 0; i < nLights; ++i) {
    const Light *light = scene->lights[i];
    int nSamples = light->nSamples;
    if (sampler) nSamples = sampler->RoundSize(nSamples);
    lightSampleOffsets[i] = LightSampleOffsets(nSamples, sample);
    bsdfSampleOffsets[i] = BSDFSampleOffsets(nSamples, sample);
}
```

Things are easier if only a single sample needs to be taken from a single light. Other than the immediate differences from this change, the only other difference is that an additional 1D sample is needed to choose which light to sample.

BSDFSampleOffsets 706
DirectLightingIntegrator::bsdfSampleOffsets 743
DirectLightingIntegrator::lightNumOffset 742
DirectLightingIntegrator::lightSampleOffsets 743
DirectLightingIntegrator::SAMPLE_ALL_UNIFORM 742
DirectLightingIntegrator::strategy 742
Light 606
LightSampleOffsets 710
Sample 343
Sampler 340
Sampler::RoundSize() 344
Scene 22
Scene::lights 23

⟨*Allocate and request samples for sampling one light*⟩ ≡ 743
```
lightSampleOffsets = new LightSampleOffsets[1];
lightSampleOffsets[0] = LightSampleOffsets(1, sample);
lightNumOffset = sample->Add1D(1);
bsdfSampleOffsets = new BSDFSampleOffsets[1];
bsdfSampleOffsets[0] = BSDFSampleOffsets(1, sample);
```

The general form of the `DirectLightingIntegrator::Li()` method is similar to that of `WhittedIntegrator::Li()`. The BSDF at the intersection point is computed, emitted radiance is added if the surface is emissive, and so on. We won't include the full implementation of `DirectLightingIntegrator::Li()` here in order to focus on its key fragment, ⟨*Compute direct lighting at hit point*⟩, which estimates the value of the integral that gives the reflected radiance.

The scattering equation from Section 5.6 says that exitant radiance $L_o(p, \omega_o)$ from a point p on a surface in direction ω_o is the sum of emitted radiance from the surface at the point plus the integral of incoming radiance over the sphere times the BSDF for each direction and a cosine term. For the `DirectLightingIntegrator`, we are only interested in incident radiance directly from light sources, which we denote by $L_d(p, \omega)$:

$$L_o(p, \omega_o) = L_e(p, \omega_o) + \int_{S^2} f(p, \omega_o, \omega_i)\, L_d(p, \omega_i)\, |\cos \theta_i|\, d\omega_i. \quad [15.1]$$

The value of $L_e(p, \omega_o)$ is found by calling the `Intersection::Le()` method at the intersection point. To estimate the integral over the sphere, we will apply Monte Carlo integration.

⟨*Compute direct lighting for* `DirectLightingIntegrator` *integrator*⟩ ≡
```
if (scene->lights.size() > 0) {
    ⟨Apply direct lighting strategy⟩
}
```

The fragment ⟨*Apply direct lighting strategy*⟩ is also not included here. It just calls one of the functions that implement the direct lighting approaches—`UniformSampleAllLights()` or `UniformSampleOneLight()`—depending on the value of the `strategy` member variable.

To understand the approaches implemented by the different strategies, first consider the part of the direct lighting equation that we're concerned with here:

$$\int_{S^2} f(p, \omega_o, \omega_i)\, L_d(p, \omega_i)\, |\cos \theta_i|\, d\omega_i.$$

This can be broken into a sum over the lights in the scene

$$\sum_{j=1}^{\text{lights}} \int_{S^2} f(p, \omega_o, \omega_i)\, L_{d(j)}(p, \omega_i)\, |\cos \theta_i|\, d\omega_i,$$

where $L_{d(j)}$ denotes incident radiance from the jth light and

$$L_d(p, \omega_i) = \sum_j L_{d(j)}(p, \omega_i).$$

BSDFSampleOffsets 706
DirectLightingIntegrator::
bsdfSampleOffsets 743
DirectLightingIntegrator::
Li() 744
DirectLightingIntegrator::
lightNumOffset 742
DirectLightingIntegrator::
lightSampleOffsets 743
Intersection::Le() 625
LightSampleOffsets 710
Sample::Add1D() 344
Scene::lights 23
UniformSampleAllLights() 745
UniformSampleOneLight() 746
WhittedIntegrator::Li() 42

One valid approach is to estimate each term of this sum individually, adding the results together. This is the most basic direct lighting strategy and is implemented in `UniformSampleAllLights()`, which we have implemented as a global function rather than a `DirectLightingIntegrator` method so that other integrators can use it as well.

Note that the number of samples for each light is *not* taken from its `nSamples` member variable but instead from the number of samples available in the `Sample` (the value of which, in turn, was returned by the earlier `Sampler::RoundSize()` call). Note also that the `lightSampleOffsets` and `bsdfSampleOffsets` parameters may be `NULL`, in which case uniform random samples are used for the lighting calculation instead.

⟨*Integrator Utility Functions*⟩ +≡
```
Spectrum UniformSampleAllLights(const Scene *scene,
        const Renderer *renderer, MemoryArena &arena, const Point &p,
        const Normal &n, const Vector &wo, float rayEpsilon,
        float time, BSDF *bsdf, const Sample *sample, RNG &rng,
        const LightSampleOffsets *lightSampleOffsets,
        const BSDFSampleOffsets *bsdfSampleOffsets) {
    Spectrum L(0.);
    for (uint32_t i = 0; i < scene->lights.size(); ++i) {
        Light *light = scene->lights[i];
        int nSamples = lightSampleOffsets ?
                       lightSampleOffsets[i].nSamples : 1;
        ⟨Estimate direct lighting from light samples 745⟩
    }
    return L;
}
```

BSDF 478
BSDFSample 705
BSDFSampleOffsets 706
BSDF_ALL 428
BSDF_SPECULAR 428
BxDFType 428
EstimateDirect() 749
Light 606
LightSample 710
LightSampleOffsets 710
MemoryArena 1015
Normal 65
Point 63
Renderer 24
RNG 1003
Sample 343
Sampler::RoundSize() 344
Scene 22
Scene::lights 23
Spectrum 263
UniformSampleAllLights() 745
Vector 57

For each light sample the `EstimateDirect()` function, to be defined shortly, computes the value of the Monte Carlo estimator for its contribution. All that has to be done here is to average the values of the estimates.

⟨*Estimate direct lighting from* `light` *samples*⟩ ≡ 745
```
Spectrum Ld(0.);
for (int j = 0; j < nSamples; ++j) {
    ⟨Find light and BSDF sample values for direct lighting estimate 746⟩
    Ld += EstimateDirect(scene, renderer, arena, light, p, n, wo,
        rayEpsilon, time, bsdf, rng, lightSample, bsdfSample,
        BxDFType(BSDF_ALL & ~BSDF_SPECULAR));
}
L += Ld / nSamples;
```

Before calling `EstimateDirect()`, this function assembles the values for the various random samples that will be used for Monte Carlo integration. The `LightSample` and `BSDFSample` structures encapsulate the collections of random values that the lights and BSDFs need for their sampling routines; their constructors take a `Sample`, the respective `SampleOffsets` types, and the sample number and extract the appropriate sample values

from the Sample structure. If the SampleOffsets types are NULL, then a reference to a random number generator is passed to the sample constructors so that they can generate uniform random samples.

⟨*Find light and BSDF sample values for direct lighting estimate*⟩ ≡ 745
```
LightSample lightSample;
BSDFSample bsdfSample;
if (lightSampleOffsets != NULL && bsdfSampleOffsets != NULL) {
    lightSample = LightSample(sample, lightSampleOffsets[i], j);
    bsdfSample = BSDFSample(sample, bsdfSampleOffsets[i], j);
}
else {
    lightSample = LightSample(rng);
    bsdfSample = BSDFSample(rng);
}
```

In a scene with a large number of lights, it may not be desirable to always compute direct lighting from all of the lights at every point that is shaded. Monte Carlo gives a way to do this that still computes the correct result on average. Consider as an example computing the expected value of the sum of two functions $E[f(x) + g(x)]$. If we randomly evaluate just one of $f(x)$ or $g(x)$ and multiply the result by two, then the expected value of the result will still be $f(x) + g(x)$. In fact, this generalizes to sums of N terms. This is a straightforward application of conditional probability; see Ross (2002, p. 102) for a proof. Here we estimate direct lighting for only one randomly chosen light and multiply the result by the number of lights to compensate.

⟨*Integrator Utility Functions*⟩ +≡
```
Spectrum UniformSampleOneLight(const Scene *scene,
        const Renderer *renderer, MemoryArena &arena, const Point &p,
        const Normal &n, const Vector &wo, float rayEpsilon, float time,
        BSDF *bsdf, const Sample *sample, RNG &rng, int lightNumOffset,
        const LightSampleOffsets *lightSampleOffset,
        const BSDFSampleOffsets *bsdfSampleOffset) {
    ⟨Randomly choose a single light to sample, light 747⟩
    ⟨Initialize light and bsdf samples for single light sample 747⟩
    return (float)nLights *
        EstimateDirect(scene, renderer, arena, light, p, n, wo,
                       rayEpsilon, time, bsdf, rng, lightSample,
                       bsdfSample, BxDFType(BSDF_ALL & ~BSDF_SPECULAR));
}
```

Which of the nLights to sample illumination from is determined using the random sample available in the 1D integrator sample lightNumOffset, if available. Although this is never the case for the DirectLightingIntegrator integrator, sometimes other integrators will call this function without such a sample available. In that case, they pass −1 for the sample offset and a uniform random number is used instead.

BSDF 478
BSDFSample 705
BSDFSampleOffsets 706
BSDF_ALL 428
BSDF_SPECULAR 428
BxDFType 428
DirectLightingIntegrator 742
EstimateDirect() 749
LightSample 710
LightSampleOffsets 710
MemoryArena 1015
Normal 65
Point 63
Renderer 24
RNG 1003
Sample 343
Scene 22
Spectrum 263
Vector 57

⟨*Randomly choose a single light to sample,* `light`⟩ ≡ 746
```
int nLights = int(scene->lights.size());
if (nLights == 0) return Spectrum(0.);
int lightNum;
if (lightNumOffset != -1)
    lightNum = Floor2Int(sample->oneD[lightNumOffset][0] * nLights);
else
    lightNum = Floor2Int(rng.RandomFloat() * nLights);
lightNum = min(lightNum, nLights-1);
Light *light = scene->lights[lightNum];
```

It's possible to be even more creative in choosing the individual light sampling probabilities than the uniform method used in `UniformSampleOneLight()`. In fact, we're free to set the probabilities any way we like, so long as we weight the result appropriately and there is a nonzero probability of sampling any light that contributes to the reflection at the point. The better a job we do at setting the probabilities, the more efficient the Monte Carlo estimator will be, and the fewer rays will be needed to lower variance to an acceptable level. (This is just the discrete instance of importance sampling.) One widely used approach is to base the sample distribution on the total power of each light. In a similar manner, we could take more than one light sample with such an approach; indeed, any number of samples can be taken in the end, so long as they are weighted appropriately. This topic is discussed further in Exercise 15.5 at the end of the chapter.

⟨*Initialize light and bsdf samples for single light sample*⟩ ≡ 746
```
LightSample lightSample;
BSDFSample bsdfSample;
if (lightSampleOffset != NULL && bsdfSampleOffset != NULL) {
    lightSample = LightSample(sample, *lightSampleOffset, 0);
    bsdfSample = BSDFSample(sample, *bsdfSampleOffset, 0);
}
else {
    lightSample = LightSample(rng);
    bsdfSample = BSDFSample(rng);
}
```

15.1.1 ESTIMATING THE DIRECT LIGHTING INTEGRAL

Having chosen a particular light to estimate direct lighting from, we need to estimate the value of the integral

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

for that light. To compute this estimate, we need to choose one or more directions ω_j and apply the Monte Carlo estimator:

$$\frac{1}{N}\sum_{j=1}^{N} \frac{f(\mathrm{p}, \omega_\mathrm{o}, \omega_j)\, L_\mathrm{d}(\mathrm{p}, \omega_j)\, |\cos\theta_j|}{p(\omega_j)}.$$

BSDFSample 705
Floor2Int() 1002
Light 606
LightSample 710
RNG::RandomFloat() 1003
Sample::oneD 344
Scene::lights 23
Spectrum 263
UniformSampleOneLight() 746

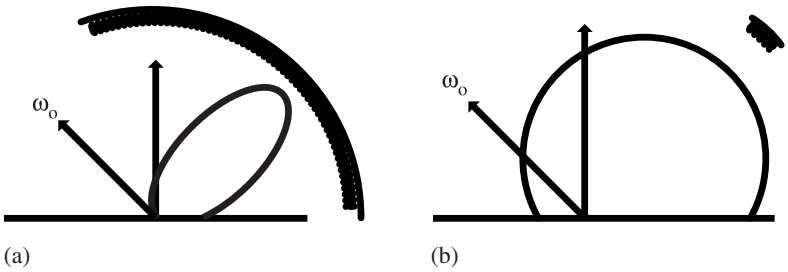


Figure 15.2: Depending on the actual distributions of the BSDF and the light source as a function of direction from the point being illuminated, one or the other may be a much more effective distribution to draw samples from for importance sampling. (a) Sampling the BSDF will generally be more effective, since illumination is coming from many directions but the BSDF's value is large for only a few of them. (b) The converse is true: sampling the BSDF in this case would select many directions along which no incident illumination was arriving.

To reduce variance, we will use importance sampling to choose the directions ω_j. Because both the BSDF and the direct radiance terms are individually complex, it can be difficult to find sampling distributions that match their product well. (However, see the "Further Reading" section as well as Exercise 15.3 at the end of this chapter for approaches for sampling their product directly.) Here we will use the BSDF's sampling distribution for some of the samples and the light's for the rest. Depending on the characteristics of each of them, one of these two sampling methods may be far more effective than the other. Therefore, we will use multiple importance sampling to reduce variance for the cases where one or the other is more effective.

Figure 15.2 shows two cases where one of the sampling methods is much better than the other. In Figure 15.2(a), the BSDF is very specular, and the light source is relatively large. Sampling the BSDF will be effective at finding directions where the integrand's value is large, while sampling the light will be less effective: most of the samples will not contribute much since the BSDF is small for most of the directions to the light source. When the light happens to sample a point in the BSDF's glossy region, there is a spike in the image because the light returns a low PDF, while the value of the integrand is relatively large. As a result, the variance is high because the sampling distribution didn't match the actual distribution of the function's values very well.

On the other hand, sometimes sampling the light is the right strategy. In Figure 15.2(b), the BSDF is less highly specular and thus nonzero over many directions, while the light is relatively small. It will be far more effective to choose points on the light to select the incident direction since the BSDF will have trouble finding directions where there is nonzero incident radiance from the light. Similarly to the first case, we would encounter substantial variance since the sampling distribution didn't match the overall function well.

By applying multiple importance sampling, not only can we use both of the two sampling methods, but we can also do so in a way that eliminates the extreme variance from these two situations, since the weighting terms from MIS reduce this variance substantially.

⟨*Integrator Utility Functions*⟩ +≡
```
Spectrum EstimateDirect(const Scene *scene, const Renderer *renderer,
        MemoryArena &arena, const Light *light, const Point &p,
        const Normal &n, const Vector &wo, float rayEpsilon, float time,
        const BSDF *bsdf, RNG &rng, const LightSample &lightSample,
        const BSDFSample &bsdfSample, BxDFType flags) {
    Spectrum Ld(0.);
    ⟨Sample light source with multiple importance sampling 749⟩
    ⟨Sample BSDF with multiple importance sampling 750⟩
    return Ld;
}
```

First, one sample is taken with the light's sampling distribution using `Sample_L()`, which also returns the light's emitted radiance and the value of the PDF for the sampled direction. Only if the light successfully samples a direction and returns nonzero emitted radiance does the function here go ahead and evaluate the BSDF for the two directions; otherwise, there's no reason to go through the computational expense. For example, a spotlight returns zero radiance for points outside its illumination cone. Then only if the BSDF's value is nonzero is a shadow ray traced to check for occlusion.

⟨*Sample light source with multiple importance sampling*⟩ ≡ 749
```
Vector wi;
float lightPdf, bsdfPdf;
VisibilityTester visibility;
Spectrum Li = light->Sample_L(p, rayEpsilon, lightSample, time,
                              &wi, &lightPdf, &visibility);
if (lightPdf > 0. && !Li.IsBlack()) {
    Spectrum f = bsdf->f(wo, wi, flags);
    if (!f.IsBlack() && visibility.Unoccluded(scene)) {
        ⟨Add light's contribution to reflected radiance 750⟩
    }
}
```

BSDF 478
BSDF::f() 481
BSDF::Pdf() 708
BSDFSample 705
BxDFType 428
Light 606
Light::Sample_L() 608
LightSample 710
MemoryArena 1015
Normal 65
Point 63
Renderer 24
RNG 1003
Scene 22
Spectrum 263
Spectrum::IsBlack() 265
Vector 57
VisibilityTester 608
VisibilityTester::
  Unoccluded() 609

Once it is known that the light is visible and radiance is arriving at the point, the value of the Monte Carlo estimator can be computed. First, radiance from the light to the illuminated point is scaled by the beam transmittance between the two points to account for attenuation due to participating media. Next, recall from Section 14.6.2 that if the light is described by a delta distribution then there is an implied delta distribution in both the emitted radiance value returned from `Sample_L()` as well as the PDF and that they are expected to cancel out when the estimator is evaluated. In this case, we must not try to apply multiple importance sampling and should compute the standard estimator instead. If this isn't a delta distribution light source, then the `BSDF::Pdf()` method is called to return the BSDF's PDF value for sampling the direction ω_i, and the MIS estimator is used, where the weight is computed here with the power heuristic.

⟨*Add light's contribution to reflected radiance*⟩ ≡ 749
```
Li *= visibility.Transmittance(scene, renderer, NULL, rng, arena);
if (light->IsDeltaLight())
    Ld += f * Li * (AbsDot(wi, n) / lightPdf);
else {
    bsdfPdf = bsdf->Pdf(wo, wi, flags);
    float weight = PowerHeuristic(1, lightPdf, 1, bsdfPdf);
    Ld += f * Li * (AbsDot(wi, n) * weight / lightPdf);
}
```

In a similar manner, a sample is now generated using the BSDF's sampling distribution. This step can be skipped if the light source is a delta distribution because, in that case, there's no chance that sampling the BSDF will give a direction that receives light from the source. If this is not the case, the BSDF can be sampled. One important detail is that the light's PDF and the multiple importance sampling weight are only computed if the BSDF component used for sampling ω_i is non specular; in the specular case, MIS shouldn't be applied since there is no chance of the light sampling the specular direction.

⟨*Sample BSDF with multiple importance sampling*⟩ ≡ 749
```
if (!light->IsDeltaLight()) {
    BxDFType sampledType;
    Spectrum f = bsdf->Sample_f(wo, &wi, bsdfSample, &bsdfPdf, flags,
                                &sampledType);
    if (!f.IsBlack() && bsdfPdf > 0.) {
        float weight = 1.f;
        if (!(sampledType & BSDF_SPECULAR)) {
            lightPdf = light->Pdf(p, wi);
            if (lightPdf == 0.)
                return Ld;
            weight = PowerHeuristic(1, bsdfPdf, 1, lightPdf);
        }
        ⟨Add light contribution from BSDF sampling 750⟩
    }
}
```

Given a direction sampled by the BSDF, we need to find out if the ray along that direction intersects this particular light source, and if so, how much radiance from the light reaches the surface. The code must account for both regular area lights, with geometry associated with them, as well as lights like the `InfiniteAreaLight` that don't have geometry but need to return their radiance for the sample ray via the `Light::Le()` method.

⟨*Add light contribution from BSDF sampling*⟩ ≡ 750
```
Intersection lightIsect;
Spectrum Li(0.f);
RayDifferential ray(p, wi, rayEpsilon, INFINITY, time);
if (scene->Intersect(ray, &lightIsect)) {
    if (lightIsect.primitive->GetAreaLight() == light)
        Li = lightIsect.Le(-wi);
}
```

AbsDot() 61
BSDF::Pdf() 708
BSDF::Sample_f() 706
BSDF_SPECULAR 428
BxDFType 428
GeometricPrimitive::GetAreaLight() 188
InfiniteAreaLight 629
INFINITY 1002
Intersection 186
Intersection::Le() 625
Intersection::primitive 186
Light::IsDeltaLight() 608
Light::Le() 631
Light::Pdf() 711
PowerHeuristic() 693
RayDifferential 69
Renderer::Transmittance() 25
Scene::Intersect() 23
Spectrum 263
Spectrum::IsBlack() 265
VisibilityTester::Transmittance() 609

```
else
    Li = light->Le(ray);
if (!Li.IsBlack()) {
    Li *= renderer->Transmittance(scene, ray, NULL, rng, arena);
    Ld += f * Li * AbsDot(wi, n) * weight / bsdfPdf;
}
```

15.2 THE LIGHT TRANSPORT EQUATION

The light transport equation (LTE) is the governing equation that describes the equilibrium distribution of radiance in a scene. It gives the total reflected radiance at a point on a surface in terms of emission from the surface, its BSDF, and the distribution of incident illumination arriving at the point. The key task of the `Integrator` objects in `pbrt` is to numerically compute a solution to the LTE to find the incident radiance arriving at the camera. For now we will consider the case where there are no participating media in the scene. Chapter 16 describes the generalizations to this process necessary for scenes that do have participating media.

The detail that makes evaluating the LTE difficult is the fact that incident radiance at a point is affected by the geometry and scattering properties of all of the objects in the scene. For example, a bright light shining on a red object may cause a reddish tint on nearby objects in the scene, or glass may focus light into caustic patterns on a tabletop. Rendering algorithms that account for this complexity are often called *global illumination* algorithms, to differentiate them from *local illumination* algorithms that use only information about the local surface properties in their shading computations.

In this section, we will first derive the LTE and describe some approaches for manipulating the equation to make it easier to solve numerically. We will then describe two generalizations of the LTE that make some of its key properties more clear and serve as the foundation for some of the advanced integrators we will implement later in this chapter.

15.2.1 BASIC DERIVATION

The light transport equation depends on the basic assumptions we have already made in choosing to use radiometry to describe light—that wave optics effects are unimportant and that the distribution of radiance in the scene is in equilibrium. To compute radiance arriving at the film along a camera ray, we would like to express the exitant radiance from the camera ray's intersection point p in the direction ω_o from p to the film sample. We will denote this radiance measurement by $L_o(p, \omega_o)$.

The key principle underlying the LTE is *energy balance*. Any change in energy has to be "charged" to some process, and we must keep track of all the energy. Since we are assuming that lighting is a linear process, the difference between the amount of energy coming in and energy going out of a system must also be equal to the difference between energy emitted and energy absorbed. This idea holds at many levels of scale. On a macro level we have conservation of power:

Integrator 740

$$\Phi_o - \Phi_i = \Phi_e - \Phi_a.$$

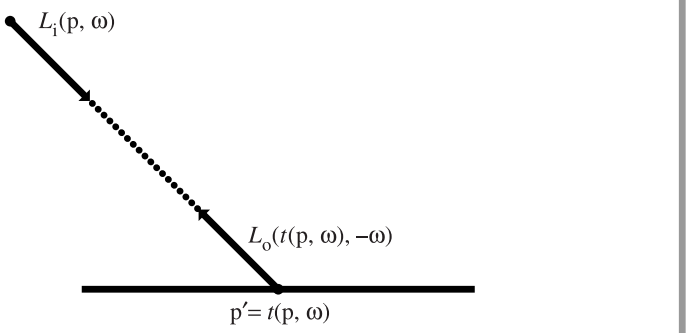


Figure 15.3: Radiance along a Ray through Free Space Is Unchanged. Therefore, to compute the incident radiance along a ray from point p in direction ω, we can find the first surface the ray intersects and compute exitant radiance in the direction $-\omega$ there. The trace operator $t(p, \omega)$ gives the point p' on the first surface that the ray (p, ω) intersects.

The difference between the power leaving an object, Φ_o, and the power entering it, Φ_i, is equal to the difference between the power it emits and the power it absorbs, $\Phi_e - \Phi_a$.

In order to enforce energy balance at a surface, exitant radiance must be equal to emitted radiance plus the fraction of incident radiance that is scattered. Emitted radiance is given by L_e, and scattered radiance is given by the scattering equation, which gives

$$L_o(p, \omega_o) = L_e(p, \omega_o) + \int_{S^2} f(p, \omega_o, \omega_i)\, L_i(p, \omega_i)\, |\cos \theta_i|\, d\omega_i.$$

Because we have assumed for now that no participating media are present, radiance is constant along rays through the scene. We can therefore relate the incident radiance at p to the outgoing radiance from another point p', as shown by Figure 15.3. If we define the *ray-casting function* $t(p, \omega)$ as a function that computes the first surface point p' intersected by a ray from p in the direction ω, we can write the incident radiance at p in terms of outgoing radiance at p':

$$L_i(p, \omega) = L_o(t(p, \omega), -\omega).$$

In case the scene is not closed, we will define the ray-casting function to return a special value Λ if the ray (p, ω) doesn't intersect any object in the scene, such that $L_o(\Lambda, \omega)$ is always zero.

Dropping the subscripts from L_o for brevity, this relationship allows us to write the LTE as

$$L(p, \omega_o) = L_e(p, \omega_o) + \int_{S^2} f(p, \omega_o, \omega_i)\, L(t(p, \omega_i), -\omega_i)\, |\cos \theta_i|\, d\omega_i. \quad [15.2]$$

The key to the above representation is that there is only *one* quantity of interest, exitant radiance from points on surfaces. Of course, it appears on both sides of the equation, so our task is still not simple, but it is certainly better. It is important to keep in mind that we were able to arrive at this equation simply by enforcing energy balance in our scene.

15.2.2 ANALYTIC SOLUTIONS TO THE LTE

The brevity of the LTE belies the fact that it is impossible to solve analytically in general. The complexity that comes from physically based BSDF models, arbitrary scene geometry, and the intricate visibility relationships between objects all conspire to mandate a numerical solution technique. Fortunately, the combination of ray-tracing algorithms and Monte Carlo integration gives a powerful pair of tools that can handle this complexity without needing to impose restrictions on various components of the LTE (e.g., requiring that all BSDFs be Lambertian or substantially limiting the geometric representations that are supported).

It is possible to find analytic solutions to the LTE in extremely simple settings. While this is of little help for general-purpose rendering, it can help with debugging the implementations of integrators. If an integrator that is supposed to solve the complete LTE doesn't compute a solution that matches an analytic solution, then clearly there is a bug in the integrator. As an example, consider the interior of a sphere where all points on the surface of the sphere have a Lambertian BRDF, $f(\mathrm{p}, \omega_\mathrm{o}, \omega_\mathrm{i}) = c$, and also emit a constant amount of radiance in all directions. We have

$$L(\mathrm{p}, \omega_\mathrm{o}) = L_\mathrm{e} + c \int_{\mathcal{H}^2(\mathbf{n})} L(t(\mathrm{p}, \omega_\mathrm{i}), -\omega_\mathrm{i}) \, |\cos \theta_\mathrm{i}| \, \mathrm{d}\omega_\mathrm{i}.$$

The outgoing radiance distribution at any point on the sphere interior must be the same as at any other point; nothing in the environment could introduce any variation among different points. Therefore, the incident radiance distribution must be the same at all points, and the cosine-weighted integral of incident radiance must be the same everywhere as well. As such, we can replace the radiance functions with constants and simplify, writing the LTE as

$$L = L_\mathrm{e} + c\pi L.$$

While we could immediately solve this equation for L, it's interesting to consider successive substitution of the right-hand side into the L term on the right-hand side. If we also replace πc with ρ_hh, the reflectance of a Lambertian surface, we have

$$\begin{aligned} L &= L_\mathrm{e} + \rho_\mathrm{hh}(L_\mathrm{e} + \rho_\mathrm{hh}(L_\mathrm{e} + \cdots \\ &= \sum_{i=0}^{\infty} L_\mathrm{e} \rho_\mathrm{hh}^i. \end{aligned}$$

In other words, exitant radiance is equal to the emitted radiance at the point plus light that has been scattered by a BSDF once after emission, plus light that has been scattered twice, and so forth.

Because $\rho_\mathrm{hh} < 1$ due to conservation of energy, the series converges and the reflected radiance at all points in all directions is

$$L = \frac{L_\mathrm{e}}{1 - \rho_\mathrm{hh}}.$$

This process of repeatedly substituting the LTE's right-hand side into the incident radiance term in the integral can be instructive in more general cases.[2] For example, the `DirectLightingIntegrator` integrator effectively computes the result of making a single substitution:

$$L(\mathrm{p}, \omega_\mathrm{o}) = L_\mathrm{e}(\mathrm{p}, \omega_\mathrm{o}) + \int_{\mathcal{S}^2} f(\mathrm{p}, \omega_\mathrm{o}, \omega_\mathrm{i})\, L_\mathrm{d}\, |\cos\theta_\mathrm{i}|\, \mathrm{d}\omega_\mathrm{i},$$

where

$$L_\mathrm{d} = L_\mathrm{e}(t(\mathrm{p}, \omega_\mathrm{i}), -\omega_\mathrm{i}) + \int_{\mathcal{S}^2} f(t(\mathrm{p}, \omega_\mathrm{i}), \omega') L(t(t(\mathrm{p}, \omega_\mathrm{i}), \omega'), -\omega') |\cos\theta'|\, \mathrm{d}\omega'$$

and then ignores the result of multiply scattered light.

Over the next few pages, we will see how performing successive substitutions in this manner and then regrouping the results expresses the LTE in a more natural way for developing rendering algorithms.

15.2.3 THE SURFACE FORM OF THE LTE

One reason why the LTE as written in Equation (15.2) is complex is that the relationship between geometric objects in the scene is implicit in the ray-tracing operator $t(\mathrm{p}, \omega)$. Making the behavior of the ray-tracing operator explicit in the integrand will shed some light on the structure of this equation. To do this, we will rewrite Equation (15.2) as an integral over *area* instead of an integral over directions on the sphere.

First, we define exitant radiance from a point p' to a point p by

$$L(\mathrm{p}' \to \mathrm{p}) = L(\mathrm{p}', \omega)$$

if p' and p are mutually visible and $\omega = \mathrm{p} \widehat{-} \mathrm{p}'$. We can also write the BSDF at p' as

$$f(\mathrm{p}'' \to \mathrm{p}' \to \mathrm{p}) = f(\mathrm{p}', \omega_\mathrm{o}, \omega_\mathrm{i})$$

where $\omega_\mathrm{i} = \mathrm{p}'' \widehat{-} \mathrm{p}'$ and $\omega_\mathrm{o} = \mathrm{p} \widehat{-} \mathrm{p}'$ (Figure 15.4).

Rewriting the terms in the LTE in this manner isn't quite enough, however. We also need to multiply by the Jacobian that relates solid angle to area in order to transform the LTE from an integral over direction to one over surface area. Recall that this is $|\cos\theta'|/r^2$.

We will combine this change-of-variables term, the original $|\cos\theta|$ term from the LTE, and also a binary visibility function V ($V = 1$ if the two points are mutually visible, and $V = 0$ otherwise) into a single geometric coupling term, $G(\mathrm{p} \leftrightarrow \mathrm{p}')$:

$$G(\mathrm{p} \leftrightarrow \mathrm{p}') = V(\mathrm{p} \leftrightarrow \mathrm{p}') \frac{|\cos\theta|\, |\cos\theta'|}{\| \mathrm{p} - \mathrm{p}' \|^2}. \qquad [15.3]$$

2 Indeed, this sort of series expansion and inversion can be used in the general case, where quantities like the BSDF are expressed in terms of general operators that map incident radiance functions to exitant radiance functions. This approach forms the foundation for applying sophisticated tools from analysis to the light transport problem. See Arvo's thesis (Arvo 1995) and Veach's thesis (Veach 1997) for further information.

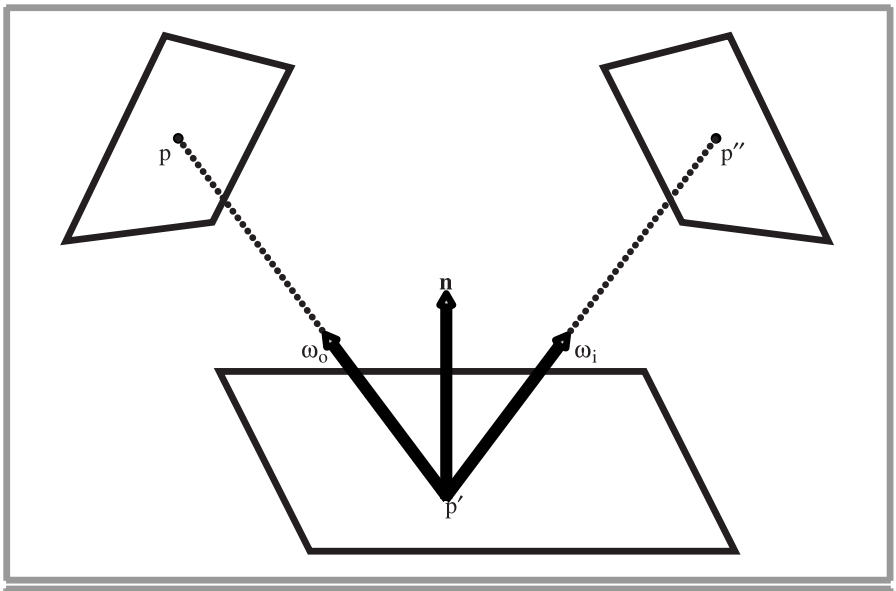


Figure 15.4: The three-point form of the light transport equation converts the integral to be over the domain of points on surfaces in the scene, rather than over directions over the sphere. It is a key transformation for deriving the path integral form of the light transport equation.

Substituting these into the light transport equation and converting to an area integral, we have

$$L(\mathrm{p}' \to \mathrm{p}) = L_\mathrm{e}(\mathrm{p}' \to \mathrm{p}) + \int_A f(\mathrm{p}'' \to \mathrm{p}' \to \mathrm{p}) L(\mathrm{p}'' \to \mathrm{p}') G(\mathrm{p}'' \leftrightarrow \mathrm{p}') \, \mathrm{d}A(\mathrm{p}''), \tag{15.4}$$

where A is all of the surfaces of the scene.

Although Equations (15.2) and (15.4) are equivalent, they represent two different ways of approaching light transport. To evaluate Equation (15.2) with Monte Carlo, we would sample a number of directions from a distribution of directions on the sphere and cast rays to evaluate the integrand. For Equation (15.4), however, we would choose a number of *points* on surfaces according to a distribution over surface area and compute the coupling between those points to evaluate the integrand, tracing rays to evaluate the visibility term $V(\mathrm{p} \leftrightarrow \mathrm{p}')$.

15.2.4 INTEGRAL OVER PATHS

To go from the area integral to a more flexible form of the LTE, a sum over light-carrying paths of different lengths, we can now start to expand the three-point light transport equation, repeatedly substituting the right-hand side of the equation into the $L(\mathrm{p}' \to \mathrm{p}')$ term inside the integral. Here are the first few terms that give incident radiance at a point p_0 from another point p_1, where p_1 is the first point on a surface along the ray from p_0 in

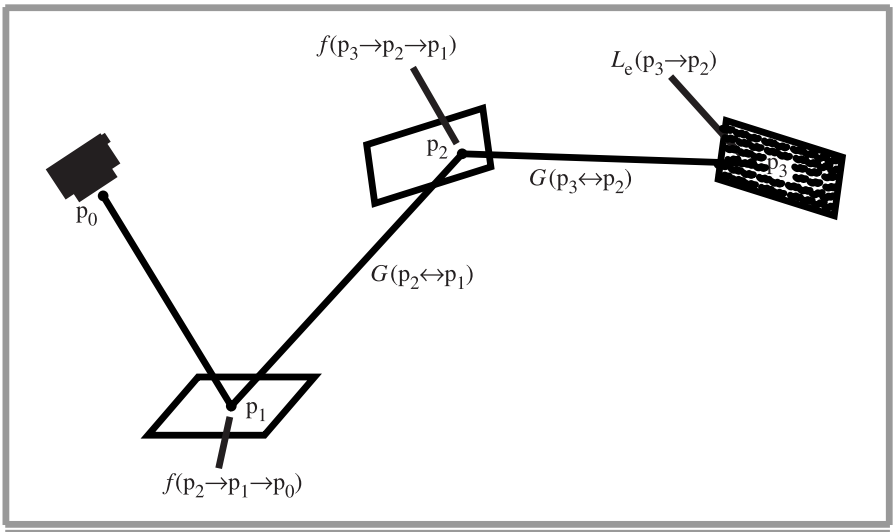


Figure 15.5: The integral over all points p_2 and p_3 on surfaces in the scene given by the light transport equation gives the total contribution of two bounce paths to radiance leaving p_1 in the direction of p_0. The components of the product in the integrand are shown here: the emitted radiance from the light, L_e; the geometric terms between vertices, G; and scattering from the BSDFs, f.

direction $p_1 - p_0$:

$$\begin{aligned} L(p_1 \to p_0) = {} & L_e(p_1 \to p_0) \\ & + \int_A L_e(p_2 \to p_1) f(p_2 \to p_1 \to p_0) G(p_2 \leftrightarrow p_1) \mathrm{d}A(p_2) \\ & + \int_A \int_A L_e(p_3 \to p_2) f(p_3 \to p_2 \to p_1) G(p_3 \leftrightarrow p_2) \\ & \qquad \times f(p_2 \to p_1 \to p_0) G(p_2 \leftrightarrow p_1) \, \mathrm{d}A(p_3) \, \mathrm{d}A(p_2) + \cdots \end{aligned}$$

Each term on the right of this equation represents a path of increasing length. For example, the third term is illustrated in Figure 15.5. This path has four vertices, connected by three segments. The total contribution of all such paths of length four (i.e., a vertex at the camera, two vertices at points on surfaces in the scene, and a vertex on a light source) is given by this term. Here, the first two vertices of the path, p_0 and p_1, are predetermined based on the camera ray and the point that it intersects, but p_2 and p_3 can vary over all points on surfaces in the scene. The integral over all such p_2 and p_3 gives the total contribution of paths of length four to radiance arriving at the camera.

This infinite sum can be written compactly as

$$L(p_1 \to p_0) = \sum_{n=1}^{\infty} P(\bar{p}_n). \tag{15.5}$$

$P(\bar{p}_n)$ gives the amount of radiance scattered over a path $\bar{p}_n$ with $n + 1$ vertices,

$$\bar{p}_n = p_0, p_1, \ldots, p_n,$$

where p_0 is on the film plane and p_n is on a light source, and

$$P(\bar{\mathrm{p}}_n) = \underbrace{\int_A \int_A \cdots \int_A}_{n-1} L_\mathrm{e}(\mathrm{p}_n \to \mathrm{p}_{n-1}) \times \left(\prod_{i=1}^{n-1} f(\mathrm{p}_{i+1} \to \mathrm{p}_i \to \mathrm{p}_{i-1}) G(\mathrm{p}_{i+1} \leftrightarrow \mathrm{p}_i) \right) \mathrm{d}A(\mathrm{p}_2) \cdots \mathrm{d}A(\mathrm{p}_n). \tag{15.6}$$

Before we move on, we will define one additional term that will be helpful in the subsequent discussion. The product of a path's BSDF and geometry terms is called the *throughput* of the path; it describes the fraction of radiance from the light source that arrives at the camera after all of the scattering at vertices between them. We will denote it by

$$T(\bar{\mathrm{p}}_n) = \prod_{i=1}^{n-1} f(\mathrm{p}_{i+1} \to \mathrm{p}_i \to \mathrm{p}_{i-1}) G(\mathrm{p}_{i+1} \leftrightarrow \mathrm{p}_i), \tag{15.7}$$

so

$$P(\bar{\mathrm{p}}_n) = \underbrace{\int_A \int_A \cdots \int_A}_{n-1} L_\mathrm{e}(\mathrm{p}_n \to \mathrm{p}_{n-1}) T(\bar{\mathrm{p}}_n) \, \mathrm{d}A(\mathrm{p}_2) \cdots \mathrm{d}A(\mathrm{p}_n).$$

Given Equation (15.5) and a particular length n, all that we need to do to compute a Monte Carlo estimate of the radiance arriving at p_0 due to paths of length n is to sample a set of vertices with an appropriate sampling density in the scene to generate a path and then to evaluate an estimate of $P(\bar{\mathrm{p}}_n)$ using those vertices. Whether we generate those vertices by starting a path from the camera, starting from the light, starting from both ends, or starting from a point in the middle is a detail that only affects how the weights for the Monte Carlo estimates are computed. We will see how this formulation leads in practice to practical light transport algorithms throughout this chapter.

15.2.5 DELTA DISTRIBUTIONS IN THE INTEGRAND

Delta functions may be present in $P(\bar{\mathrm{p}}_i)$ terms due to both BSDF components described by delta distributions as well as certain types of light sources (e.g., point lights and directional lights). These distributions need to be handled explicitly by the light transport algorithm if present. For example, it is impossible to randomly choose an outgoing direction from a point on a surface that would intersect a point light source; instead, it is necessary to explicitly choose the single direction from the point to the light source if we want to be able to include its contribution. (The same is true for sampling BSDFs with delta components.) While handling this case introduces some additional complexity to the integrators, it is generally welcome because it reduces the dimensionality of the integral to be evaluated, turning parts of it into a plain sum.

For example, consider the direct illumination term, $P(\bar{\mathrm{p}}_2)$, in a scene with a single point light source at point $\mathrm{p}_{\mathrm{light}}$ described by a delta distribution:

$$\begin{aligned} P(\bar{\mathrm{p}}_2) &= \int_A L_{\mathrm{e}}(\mathrm{p}_2 \to \mathrm{p}_1)\, f(\mathrm{p}_2 \to \mathrm{p}_1 \to \mathrm{p}_0)\, G(\mathrm{p}_2 \leftrightarrow \mathrm{p}_1)\, \mathrm{d}A(\mathrm{p}_2) \\ &= \frac{\delta(\mathrm{p}_{\mathrm{light}} - \mathrm{p}_2) L_{\mathrm{e}}(\mathrm{p}_{\mathrm{light}} \to \mathrm{p}_1)}{p(\mathrm{p}_{\mathrm{light}})} f(\mathrm{p}_2 \to \mathrm{p}_1 \to \mathrm{p}_0) G(\mathrm{p}_2 \leftrightarrow \mathrm{p}_1). \end{aligned}$$

In other words, p_2 must be the same as the light's position in the scene; the delta distribution in the numerator cancels out due to an implicit delta distribution in $p(\mathrm{p}_{\mathrm{light}})$ (recall the discussion of sampling delta distributions in Section 14.5.4), and we are left with terms that can be evaluated directly, with no need for Monte Carlo. An analogous situation holds for BSDFs with delta distributions in the path throughput $T(\bar{\mathrm{p}}_n)$; each one eliminates an integral over area from the estimate to be computed.

15.2.6 PARTITIONING THE INTEGRAND

Many rendering algorithms have been developed that are particularly good at solving the LTE under some conditions, but don't work well (or at all) under others. For example, the Whitted integrator only handles specular reflection from delta BSDFs and ignores multiply scattered light from diffuse and glossy BSDFs, and the irradiance caching technique described later in this chapter handles scattering from diffuse surfaces but would introduce significant error if used for glossy or specular reflection.

Because we would like to be able to derive correct light transport algorithms that account for all possible modes of scattering without ignoring any contributions and without double-counting others, it is important to carefully account for which parts of the LTE a particular solution method accounts for. A nice way of approaching this problem is to partition the LTE in various ways. For example, we might expand the sum over paths to

$$L(\mathrm{p}_1 \to \mathrm{p}_0) = P(\bar{\mathrm{p}}_1) + P(\bar{\mathrm{p}}_2) + \sum_{i=3}^{\infty} P(\bar{\mathrm{p}}_i),$$

where the first term is trivially evaluated by computing the emitted radiance at p_1, the second term is solved with an accurate direct lighting solution technique, but the remaining terms in the sum are handled with a faster but less accurate approach. If the contribution of these additional terms to the total reflected radiance is relatively small for the scene we're rendering, this may be a reasonable approach to take. The only detail is that it is important to be careful to ignore $P(\bar{\mathrm{p}}_1)$ and $P(\bar{\mathrm{p}}_2)$ with the algorithm that handles $P(\bar{\mathrm{p}}_3)$ and beyond (and similarly with the other terms).

It is also useful to partition individual $P(\bar{\mathrm{p}}_n)$ terms. For example, we might want to split the emission term into emission from small light sources, $L_{\mathrm{e,s}}$, and emission from large light sources, $L_{\mathrm{e,l}}$, giving us two separate integrals to estimate:

$$\begin{aligned} P(\bar{\mathrm{p}}_n) &= \int_{A^{n-1}} (L_{\mathrm{e,s}}(\mathrm{p}_n \to \mathrm{p}_{n-1}) + L_{\mathrm{e,l}}(\mathrm{p}_n \to \mathrm{p}_{n-1}))\, T(\bar{\mathrm{p}}_n)\, \mathrm{d}A(\mathrm{p}_2) \cdots \mathrm{d}A(\mathrm{p}_n) \\ &= \int_{A^n} L_{\mathrm{e,s}}(\mathrm{p}_n \to \mathrm{p}_{n-1})\, T(\bar{\mathrm{p}}_n)\, \mathrm{d}A(\mathrm{p}_2) \cdots \mathrm{d}A(\mathrm{p}_n) \\ &\quad + \int_{A^n} L_{\mathrm{e,l}}(\mathrm{p}_n \to \mathrm{p}_{n-1})\, T(\bar{\mathrm{p}}_n)\, \mathrm{d}A(\mathrm{p}_2) \cdots \mathrm{d}A(\mathrm{p}_n). \end{aligned}$$

The two integrals can be evaluated independently, possibly using completely different algorithms or different numbers of samples, selected in a way that handles the different conditions well. As long as the estimate of the $L_{\mathrm{e,s}}$ integral ignores any emission from large lights, the estimate of the $L_{\mathrm{e,l}}$ integral ignores emission from small lights, and all lights are categorized as either "large" or "small," the correct result is computed in the end.

Finally, the BSDF terms can be partitioned as well (in fact, this application was the reason why BSDF categorization with `BxDFType` values was introduced in Section 8.1). For example, if f_Δ denotes components of the BSDF described by delta distributions and $f_{\neg\Delta}$ denotes the remaining components,

$$\begin{aligned} P(\bar{\mathrm{p}}_n) = \int_{A^{n-1}} & L_{\mathrm{e}}(\mathrm{p}_n \to \mathrm{p}_{n-1}) \\ & \times \prod_{i=1}^{n-1} \left(f_\Delta(\mathrm{p}_{i+1} \to \mathrm{p}_i \to \mathrm{p}_{i-1}) + f_{\neg\Delta}(\mathrm{p}_{i+1} \to \mathrm{p}_i \to \mathrm{p}_{i-1}) \right) \\ & \qquad \times G(\mathrm{p}_{i+1} \leftrightarrow \mathrm{p}_i) \mathrm{d}A(\mathrm{p}_2) \cdots \mathrm{d}A(\mathrm{p}_n). \end{aligned}$$

Note that because there are $i - 1$ BSDF terms in the product, it is important to be careful not to count only terms with only f_Δ components or only $f_{\neg\Delta}$ components; all of the terms like $f_\Delta f_{\neg\Delta} f_{\neg\Delta}$ must be accounted for as well if a partitioning scheme like this is used.

15.2.7 THE MEASUREMENT EQUATION AND IMPORTANCE

In light of the path integral form of the LTE, it's useful to go back and formally describe the quantity that is being estimated as we compute pixel values for the image. Doing so will help us be able to apply the LTE to a wider set of problems than just computing 2D images (for example, to precomputing scattered radiance distributions at the vertices of a polygonal model, giving a theoretical grounding for the precomputed light transport methods that will be introduced in Chapter 17). Furthermore, this process leads us to a key theoretical mechanism for understanding particle tracing and the photon mapping algorithm that will be described in Section 15.6.

The *measurement equation* describes the value of an abstract measurement that is found by integrating over some set of rays carrying radiance. For example, when computing the value of a pixel in the image, we want to integrate over rays starting in the neighborhood of the pixel, with contributions weighted by the image reconstruction filter. Ignoring depth of field for now (so that each point on the film plane corresponds to a single outgoing direction from the camera), we can write the pixel's value as an integral over points on the film plane of a weighting function times the incident radiance along the

corresponding camera rays:

$$\begin{aligned} I_j &= \int_{A_{\text{film}}} \int_{S^2} W_e(p_{\text{film}}, \omega)\, L_i(p_{\text{film}}, \omega)\, |\cos\theta|\, dA(p_{\text{film}})\, d\omega \\ &= \int_{A_{\text{film}}} \int_{A} W_e(p_0 \to p_1)\, L(p_1 \to p_0)\, G(p_0 \leftrightarrow p_1)\, dA(p_0)\, dA(p_1), \end{aligned}$$

where I_j is the measurement for the jth pixel and p_0 is a point on the film. In this setting, the $W_e(p_0 \to p_1)$ term is the product of the filter function around the pixel, f_j, and a delta function that selects the appropriate camera ray direction of the sample from p_0, $\omega_{\text{camera}}(p_0)$:

$$W_e(p_0 \to p_1) = f_j(p_0)\, \delta(t(p_0, \omega_{\text{camera}}(p_0)) - p_1).$$

This formulation may initially seem gratuitously complex, but it leads us to an important insight. If we expand the $P(\bar{p}_n)$ terms of the LTE sum, we have

$$\begin{aligned} I_j &= \int_{A_{\text{film}}} \int_{A} W_e(p_0 \to p_1)\, L(p_1 \to p_0)\, G(p_0 \leftrightarrow p_1)\, dA(p_0)\, dA(p_1) \\ &= \sum_i \int_A \int_A W_e(p_0 \to p_1)\, P(\bar{p}_i)\, G(p_0 \leftrightarrow p_1)\, dA(p_0)\, dA(p_1) \\ &= \sum_i \int_A \cdots \int_A W_e(p_0 \to p_1)\, T(\bar{p}_i)\, L_e(p_{i+1} \to p_i)\, G(p_0 \leftrightarrow p_1)\, dA(p_0) \cdots dA(p_i). \end{aligned}$$

A nice symmetry between the emitted radiance from light sources (L_e) and the contribution of a sample on the film to the pixel measurement (W_e) has become apparent. The implications of this symmetry are important: it says that we can think of the rendering process in two different ways: Light could be emitted from light sources, bounce around the scene, and arrive at a sensor where W_e describes its contribution to the measurement. Alternatively, we can think of some quantity being emitted from the sensor, bouncing around the scene, and making a contribution when it hits a light source. Either intuition is equally valid.

The value described by the $W_e(p_0 \to p_1)$ term is known as the *importance* for the ray between p_0 and p_1 in the scene. When the measurement equation is used to compute pixel measurements, the importance will often be partially or fully described by delta distributions, as it was in the previous example. Many other types of measurement besides image formation can be described by appropriately constructed importance functions, and thus the formalisms described here can be used to show how the integral over paths described by the measurement equation is the integral that must be estimated to compute them. We will make use of these ideas when describing the photon mapping algorithm later in this chapter.

15.3 PATH TRACING

Now that we have derived the path integral form of the light transport equation, we'll show how it can be used to derive the *path-tracing* light transport algorithm and will present a path-tracing integrator. Figure 15.6 compares images of a scene rendered with

(a)

(b)

Figure 15.6: San Miguel Scene Rendered with Path Tracing. (a) Rendered with path tracing with 1024 samples per pixel. (b) Rendered with just 8 samples per pixel, giving the characteristic grainy noise that is the hallmark of variance. The disadvantage of path tracing is that a large number of samples may have to be taken in order to reduce variance to an acceptable level. *(Model courtesy of Guillermo M. Leal Llaguno.)*

different numbers of pixel samples using the path-tracing integrator. In general, hundreds or thousands of samples per pixel may be necessary for high-quality results—potentially a substantial computational expense.

Path tracing was the first general-purpose unbiased Monte Carlo light transport algorithm used in graphics. Kajiya (1986) introduced it in the same paper that first described the light transport equation. Path tracing incrementally generates paths of scattering events starting at the camera and ending at light sources in the scene. One way to think of it is as an extension of Whitted's method to include both delta distribution and nondelta BSDFs and light sources, rather than just accounting for the delta terms.

Although it is slightly easier to derive path tracing directly from the basic light transport equation, we will instead approach it from the path integral form, which helps build understanding of the path integral equation and will make the generalization to bidirectional path tracing easier to understand. Bidirectional path tracing is a technique where paths are generated starting from the lights as well as from the camera; it is discussed (but not implemented) at the end of this section. (See also Section 15.7.4, which has a simple bidirectional path-tracing implementation for the `MetropolisRenderer`.)

15.3.1 OVERVIEW

Given the path integral form of the LTE, we would like to estimate the value of the exitant radiance from the camera ray's intersection point p_1,

$$L(\mathrm{p}_1 \rightarrow \mathrm{p}_0) = \sum_{i=1}^{\infty} P(\bar{\mathrm{p}}_i),$$

for a given camera ray from p_0 that first intersects the scene at p_1. We have two problems that must be solved in order to compute this estimate:

1. How do we estimate the value of the sum of the infinite number of $P(\bar{\mathrm{p}}_i)$ terms with a finite amount of computation?
2. Given a particular $P(\bar{\mathrm{p}}_i)$ term, how do we generate one or more paths $\bar{\mathrm{p}}$ in order to compute a Monte Carlo estimate of its multidimensional integral?

For path tracing, we can take advantage of the fact that, for physically valid scenes, paths with more vertices scatter less light than paths with fewer vertices overall (this isn't necessarily true for any particular pair of paths, just in the aggregate). This is a natural consequence of conservation of energy in BSDFs. Therefore, we will always estimate the first few terms $P(\bar{\mathrm{p}}_i)$ and will then start to apply Russian roulette to stop sampling after a finite number of terms without introducing bias. Recall from Section 14.1 that Russian roulette allows us to probabilistically stop computing terms in a sum so long as we reweight the terms that are not terminated. For example, if we always computed estimates of $P(\bar{\mathrm{p}}_1)$, $P(\bar{\mathrm{p}}_2)$, and $P(\bar{\mathrm{p}}_3)$ but stopped without computing more terms with probability q, then an unbiased estimate of the sum would be

$$P(\bar{\mathrm{p}}_1) + P(\bar{\mathrm{p}}_2) + P(\bar{\mathrm{p}}_3) + \frac{1}{1-q} \sum_{i=4}^{\infty} P(\bar{\mathrm{p}}_i).$$

Using Russian roulette in this way doesn't solve the problem of needing to evaluate an infinite sum, but has pushed it a bit farther out.

If we take this idea a step further and instead randomly consider terminating evaluation of the sum at each term with probability q_i,

$$\frac{1}{1-q_1}\left(P(\bar{\mathrm{p}}_1) + \frac{1}{1-q_2}\left(P(\bar{\mathrm{p}}_2) + \frac{1}{1-q_3}\left(P(\bar{\mathrm{p}}_3) + \cdots,\right.\right.\right.$$

we will eventually stop continued evaluation of the sum. Yet, because for any particular value of i there is greater than zero probability of evaluating the term $P(\bar{\mathrm{p}}_i)$ and because it will be weighted appropriately if we do evaluate it, the final result is an unbiased estimate of the sum.

15.3.2 PATH SAMPLING

Given this method for evaluating only a finite number of terms of the infinite sum, we also need a way to estimate the contribution of a particular term $P(\bar{\mathrm{p}}_i)$. We need $i + 1$ vertices to specify the path, where the last vertex p_i is on a light source and the first vertex p_0 is determined by the camera ray's first intersection point (Figure 15.7). Looking at the form of $P(\bar{\mathrm{p}}_i)$, a multiple integral over surface area of objects in the scene, the most natural thing to do is to sample vertices p_i according to the surface area of objects in the scene, such that it's equally probable to sample any particular point on an object in the scene for p_i as any other point. (We don't actually use this approach in the `PathIntegrator` implementation for reasons that will be described later, but this sampling technique could possibly be used to improve the efficiency of our basic implementation and helps to clarify the meaning of the path integral LTE.)

We could define a discrete probability over the n objects in the scene. If each has surface area A_i, then the probability of sampling a path vertex on the surface of the ith object

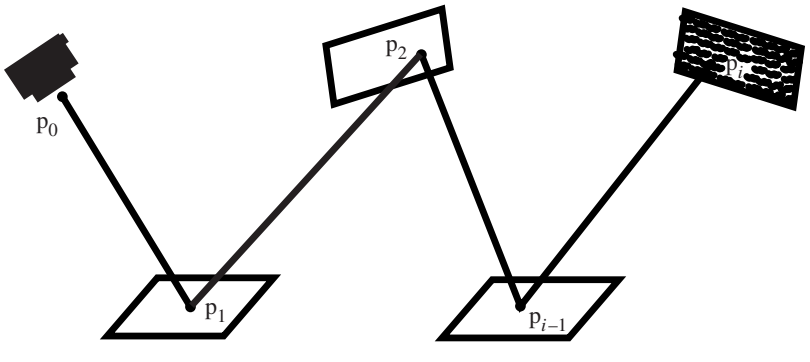


Figure 15.7: A path $\bar{\mathrm{p}}_i$ of $i + 1$ vertices from the camera at p, intersecting a series of positions on surfaces in the scene, to a point on the light p_i. Scattering according to the BSDF occurs at each path vertex from p_i to p_{i-1} such that the radiance arriving at the camera due to this path is given by the product of the path throughput $T(\bar{\mathrm{p}}_i)$ times the emitted radiance from the light.

should be

$$p_i = \frac{A_i}{\sum_j A_j}.$$

Then, given a method to sample a point on the ith object with uniform probability, the PDF for sampling any particular point on object i is $1/A_i$. Thus, the overall probability density for sampling the point is

$$\frac{A_i}{\sum_j A_j} \frac{1}{A_i}.$$

And all samples p_i have the same PDF value:

$$p_A(\mathrm{p}_i) = \frac{1}{\sum_j A_j}.$$

It's reassuring that they all have the same weight, since our intent was to choose among all points on surfaces in the scene with equal probability.

Given the set of vertices $\mathrm{p}_0, \mathrm{p}_1, \ldots, \mathrm{p}_{i-1}$ sampled in this manner, we can then sample the last vertex p_i on a light source in the scene, defining its PDF in the same way. Although we could use the same technique used for sampling path vertices to sample points on lights, this would lead to high variance, since for all of the paths where p_i wasn't on the surface of an emitter, the path would have zero value. The expected value would still be the correct value of the integral, but convergence would be extremely slow. A better approach is to sample over the areas of only the emitting objects with probabilities updated accordingly. Given a complete path, we have all of the information we need to compute the estimate of $P(\bar{\mathrm{p}}_i)$; it's just a matter of evaluating each of the terms.

It's easy to be more creative about how we set the sampling probabilities with this general approach. For example, if we knew that indirect illumination from a few objects contributed to most of the lighting in the scene, we could assign a higher probability to generating path vertices p_i on those objects, updating the sample weights appropriately.

However, there are two interrelated problems with sampling paths in this manner. The first can lead to high variance, while the second can lead to incorrect results. The first problem is that many of the paths will have no contribution if they have pairs of adjacent vertices that are not mutually visible. Consider applying this area sampling method in a complex building model: adjacent vertices in the path will almost always have a wall or two between them, giving no contribution for the path and excessive variance in the estimate.

The second problem is that if the integrand has delta functions in it (e.g., a point light source or a perfectly specular BSDF), this sampling technique will never be able to choose path vertices such that the delta distributions are nonzero. And, even if there aren't delta distributions, as the BSDFs become increasingly glossy almost all of the paths will have low contributions since the points in $f(\mathrm{p}_{i+1} \to \mathrm{p}_i \to \mathrm{p}_{i-1})$ will cause the BSDF to have a small or zero value and again we will suffer from high variance. In a similar manner, small area light sources can also be sources of variance if not sampled explicitly.

15.3.3 INCREMENTAL PATH CONSTRUCTION

A solution that solves both of these problems is to construct the path incrementally, starting from the vertex at the camera p_0. At each vertex, the BSDF is sampled to generate a new direction; the next vertex p_{i+1} is found by tracing a ray from p_i in the sampled direction and finding the closest intersection. We are effectively trying to find a path with a large overall contribution by making a series of choices that find directions with important local contributions. While one can imagine situations where this approach could be ineffective, it is generally a good strategy.

Because this approach constructs the path by sampling BSDFs according to solid angle, and because the path integral LTE is an integral over surface area in the scene, we need to apply the correction to convert from the probability density according to solid angle p_ω to a density according to area p_A (recall Section 5.5):

$$p_A = p_\omega \frac{|\cos \theta_i|}{\|\mathrm{p}_i - \mathrm{p}_{i+1}\|^2}.$$

This correction causes some of the terms of the geometric term $G(\mathrm{p}_i \leftrightarrow \mathrm{p}_{i+1})$ to cancel out of $P(\bar{\mathrm{p}}_i)$. Furthermore, we already know that p_i and p_{i+1} must be mutually visible since we traced a ray to find p_{i+1}, so the visibility term is trivially one. Therefore, if we use this sampling technique but we still sample the last vertex p_i from some distribution over the surfaces of light sources $p_A(\mathrm{p}_i)$, the value of the Monte Carlo estimate for a path is

$$\frac{L_\mathrm{e}(\mathrm{p}_i \to \mathrm{p}_{i-1}) f(\mathrm{p}_i \to \mathrm{p}_{i-1} \to \mathrm{p}_{i-2}) |\cos \theta_{\mathrm{i}-1}|}{p_A(\mathrm{p}_i)} \left(\prod_{j=1}^{i-2} \frac{f(\mathrm{p}_{j+1} \to \mathrm{p}_j \to \mathrm{p}_{j-1}) |\cos \theta_j|}{p_\omega(\mathrm{p}_{j+1} - \mathrm{p}_j)} \right). \tag{15.8}$$

15.3.4 IMPLEMENTATION

Our path-tracing implementation computes an estimate of the sum of path contributions $P(\bar{\mathrm{p}}_i)$ using the approach described in the previous subsection. Starting at the first intersection of the camera ray with the scene geometry, p_1, it incrementally samples path vertices by sampling from the BSDF's sampling distribution at the current vertex and tracing a ray to the next vertex. To find the last vertex of a particular path, p_i, which must be on a light source in the scene, it uses the multiple importance sampling–based direct lighting code that was developed for the direct lighting integrator. By using the multiple importance sampling weights instead of $p_A(\mathrm{p}_i)$ to compute the estimate as described earlier, we have lower variance in the result for cases where sampling the BSDF would have been a better way to find a point on the light.

Another small difference is that as the estimates of the path contribution terms $P(\bar{\mathrm{p}}_i)$ are being evaluated, the vertices of the previous path of length $i - 1$ (everything except the vertex on the emitter) are reused as a starting point when constructing the path of length i. This means that it is only necessary to trace one more ray to construct the new path, rather than i rays as we would if we started from scratch. Reusing paths in this manner does introduce correlation among all of the $P(\bar{\mathrm{p}}_i)$ terms in the sum, which slightly reduces the quality of the result, although in practice this is more than made up for by the improved overall efficiency due to tracing fewer rays.

⟨*PathIntegrator Declarations*⟩ ≡
```
class PathIntegrator : public SurfaceIntegrator {
public:
    ⟨PathIntegrator Public Methods 766⟩
private:
    ⟨PathIntegrator Private Data 766⟩
};
```

Although Russian roulette is used here to terminate path sampling in the manner described earlier, the integrator also supports a maximum depth. It can be set to a large number if only Russian roulette should be used:

⟨*PathIntegrator Public Methods*⟩ ≡ 766
```
PathIntegrator(int md) { maxDepth = md; }
```

⟨*PathIntegrator Private Data*⟩ ≡ 766
```
int maxDepth;
```

The integrator uses samples from the `Sampler` for sampling at the first `SAMPLE_DEPTH` vertices of the path. After the first few bounces, the advantages of well-distributed sample points are greatly reduced, and it switches to using uniform random numbers. The integrator needs light and BSDF samples for multiple importance sampling for the direct lighting calculation at each vertex of the path as well as a second set of BSDF samples for sampling directions when generating the outgoing direction for finding the next vertex of the path.

⟨*PathIntegrator Method Definitions*⟩ ≡
```
void PathIntegrator::RequestSamples(Sampler *sampler, Sample *sample,
                                    const Scene *scene) {
    for (int i = 0; i < SAMPLE_DEPTH; ++i) {
        lightSampleOffsets[i] = LightSampleOffsets(1, sample);
        lightNumOffset[i] = sample->Add1D(1);
        bsdfSampleOffsets[i] = BSDFSampleOffsets(1, sample);
        pathSampleOffsets[i] = BSDFSampleOffsets(1, sample);
    }
}
```

⟨*PathIntegrator Private Data*⟩ +≡ 766
```
#define SAMPLE_DEPTH 3
LightSampleOffsets lightSampleOffsets[SAMPLE_DEPTH];
int lightNumOffset[SAMPLE_DEPTH];
BSDFSampleOffsets bsdfSampleOffsets[SAMPLE_DEPTH];
BSDFSampleOffsets pathSampleOffsets[SAMPLE_DEPTH];
```

Each time through the `for` loop of the integrator, the next vertex of the path is found by intersecting the current ray with the scene geometry and computing the contribution of the path to the overall radiance value with the direct lighting code. A new direction is then chosen by sampling from the BSDF's distribution at the last vertex of the path. After a few vertices have been sampled, Russian roulette is used to randomly terminate the path.

BSDFSampleOffsets 706
LightSampleOffsets 710
PathIntegrator 766
PathIntegrator::
bsdfSampleOffsets 766
PathIntegrator::
lightNumOffset 766
PathIntegrator::
lightSampleOffsets 766
PathIntegrator::
pathSampleOffsets 766
Sample 343
Sample::Add1D() 344
Sampler 340
Scene 22
SurfaceIntegrator 740

⟨*PathIntegrator Method Definitions*⟩ +≡
```
Spectrum PathIntegrator::Li(const Scene *scene, const Renderer *renderer,
        const RayDifferential &r, const Intersection &isect,
        const Sample *sample, RNG &rng, MemoryArena &arena) const {
    ⟨Declare common path integration variables 767⟩
    for (int bounces = 0; ; ++bounces) {
        ⟨Possibly add emitted light at path vertex 768⟩
        ⟨Sample illumination from lights to find path contribution 768⟩
        ⟨Sample BSDF to get new path direction 768⟩
        ⟨Possibly terminate the path 769⟩
        ⟨Find next vertex of path 769⟩
    }
    return L;
}
```

A number of variables record the current state of the path. `pathThroughput` holds the product of the BSDF values and cosine terms for the vertices generated so far, divided by their respective sampling PDFs,

$$\prod_{j=1}^{i-2} \frac{f(\mathrm{p}_{j+1} \to \mathrm{p}_j \to \mathrm{p}_{j-1})|\cos\theta_j|}{p_\omega(\mathrm{p}_{j+1} - \mathrm{p}_j)}.$$

Thus, the product of `pathThroughput` with scattered light from direct lighting at the final vertex of the path gives the contribution for that overall path. One advantage of this approach is that there is no need to store the positions and BSDFs of all of the vertices of the path, only the last one.

`L` holds the radiance value from the running total of $\sum P(\bar{\mathrm{p}}_i)$, `ray` holds the next ray to be traced to extend the path one more vertex, and `specularBounce` records if the last outgoing path direction sampled was due to specular reflection; the need to track this will be explained shortly.

Finally, `isectp` points to the `Intersection` for the most recently added vertex in the path. The first time through the `for` loop, it points to `isect`, which was passed in to the `Li()` method for the first intersection of the ray with scene geometry. Information about subsequent intersections is stored in `localIsect`; `isectp` is updated to point to this after the first intersection is processed.

⟨*Declare common path integration variables*⟩ ≡ 767
```
Spectrum pathThroughput = 1., L = 0.;
RayDifferential ray(r);
bool specularBounce = false;
Intersection localIsect;
const Intersection *isectp = &isect;
```

Intersection 186
MemoryArena 1015
PathIntegrator 766
RayDifferential 69
Renderer 24
RNG 1003
Sample 343
Scene 22
Spectrum 263

If the ray hits an object that is emissive, the emission is usually ignored, since the previous path vertex already did a direct lighting computation that was responsible for all direct lighting for paths of this length. There are two exceptions to this: The first is at the initial intersection point of camera rays, since this is the only opportunity to include emission from directly visible objects. The second exception happens when the sampled direction

from the last path vertex was from a specular BSDF component. EstimateDirect() deliberately omits the effect of specular reflection in the direct lighting computation with its default parameters; therefore, if the last bounce was due to specular reflection, any emission at the intersection point must be included here.

⟨*Possibly add emitted light at path vertex*⟩ ≡ 767
```
if (bounces == 0 || specularBounce)
    L += pathThroughput * isectp->Le(-ray.d);
```

The direct lighting computation uses the UniformSampleOneLight() function, which gives an estimate of the exitant radiance from direct lighting at the vertex at the end of the current path. Scaling this value by the running product of the path contribution gives its overall contribution to the total radiance estimate.

⟨*Sample illumination from lights to find path contribution*⟩ ≡ 767
```
BSDF *bsdf = isectp->GetBSDF(ray, arena);
const Point &p = bsdf->dgShading.p;
const Normal &n = bsdf->dgShading.nn;
Vector wo = -ray.d;
if (bounces < SAMPLE_DEPTH)
    L += pathThroughput *
         UniformSampleOneLight(scene, renderer, arena, p, n, wo,
             isectp->rayEpsilon, ray.time, bsdf, sample, rng,
             lightNumOffset[bounces], &lightSampleOffsets[bounces],
             &bsdfSampleOffsets[bounces]);
else
    L += pathThroughput *
         UniformSampleOneLight(scene, renderer, arena, p, n, wo,
             isectp->rayEpsilon, ray.time, bsdf, sample, rng);
```

Now it is necessary to sample the BSDF at the vertex at the end of the current path to get an outgoing direction for the next ray to trace. The ⟨*Get* outgoingBSDFSample *for sampling new path direction*⟩ fragment initializes a BSDFSample from sample if the current path length is less than the value of SAMPLE_DEPTH, or uses RNG::RandomFloat() otherwise. The integrator updates the path throughput as described earlier and initializes ray with the ray to be traced to find the next vertex in the next iteration of the for loop.

⟨*Sample BSDF to get new path direction*⟩ ≡ 767
```
⟨Get outgoingBSDFSample for sampling new path direction 769⟩
Vector wi;
float pdf;
BxDFType flags;
Spectrum f = bsdf->Sample_f(wo, &wi, outgoingBSDFSample, &pdf,
                            BSDF_ALL, &flags);
if (f.IsBlack() || pdf == 0.)
    break;
specularBounce = (flags & BSDF_SPECULAR) != 0;
pathThroughput *= f * AbsDot(wi, n) / pdf;
ray = RayDifferential(p, wi, ray, isectp->rayEpsilon);
```

AbsDot() 61
BSDF 478
BSDF::dgShading 479
BSDF::Sample_f() 706
BSDF_ALL 428
BSDF_SPECULAR 428
BxDFType 428
DifferentialGeometry::nn 102
DifferentialGeometry::p 102
EstimateDirect() 749
Intersection::GetBSDF() 484
Intersection::Le() 625
Normal 65
PathIntegrator::
bsdfSampleOffsets 766
PathIntegrator::
lightNumOffset 766
PathIntegrator::
lightSampleOffsets 766
Point 63
Ray::d 67
RayDifferential 69
RNG::RandomFloat() 1003
Spectrum 263
Spectrum::IsBlack() 265
UniformSampleOneLight() 746
Vector 57

⟨*Get* `outgoingBSDFSample` *for sampling new path direction*⟩ ≡ 768
```
BSDFSample outgoingBSDFSample;
if (bounces < SAMPLE_DEPTH)
    outgoingBSDFSample = BSDFSample(sample, pathSampleOffsets[bounces],
                                    0);
else
    outgoingBSDFSample = BSDFSample(rng);
```

Path termination kicks in after a few bounces, with termination probability set with a fixed minimum or based on the path throughput, whichever is lower. In general, it's worth having a higher probability of terminating low-contributing paths, since they have relatively less impact on the final image. If the path isn't terminated, `pathThroughput` is updated with the Russian roulette weight and all subsequent $P(\bar{\mathrm{p}}_i)$ terms will be appropriately affected by it.

⟨*Possibly terminate the path*⟩ ≡ 767
```
if (bounces > 3) {
    float continueProbability = min(.5f, pathThroughput.y());
    if (rng.RandomFloat() > continueProbability)
        break;
    pathThroughput /= continueProbability;
}
if (bounces == maxDepth)
    break;
```

The last task in the loop is to find the next vertex of the path by tracing a ray in the sampled outgoing direction. If no intersection is found, the ray has escaped the scene and the path is terminated; if the sample from the last path vertex was due to specular reflection or transmission, then the incident illumination from light sources that aren't represented by geometry in the scene must be included here. No direct lighting computation is done when a specular BSDF sample is taken, so if the ray misses scene geometry, we still need to account for infinite area light sources—if their contribution wasn't added here, infinite area light sources wouldn't be visible in surfaces with perfect specular reflection.

In the usual case where the ray does intersect scene geometry, the path throughput is updated to account for attenuation along the new segment due to participating media and the `isectp` pointer is updated to ensure that it points at the `Intersection` for the geometric intersection just found.

⟨*Find next vertex of path*⟩ ≡ 767
```
if (!scene->Intersect(ray, &localIsect)) {
    if (specularBounce)
        for (uint32_t i = 0; i < scene->lights.size(); ++i)
            L += pathThroughput * scene->lights[i]->Le(ray);
    break;
}
if (bounces > 1)
    pathThroughput *= renderer->Transmittance(scene, ray, NULL, rng, arena);
isectp = &localIsect;
```

BSDFSample 705
Intersection 186
Light::Le() 631
PathIntegrator::
pathSampleOffsets 766
Renderer::Transmittance() 25
RNG::RandomFloat() 1003
Scene::Intersect() 23
Scene::lights 23
Spectrum::y() 273

★ 15.3.5 BIDIRECTIONAL PATH TRACING

The path-tracing algorithm described in this section was the first general light transport algorithm in graphics, handling both a wide variety of geometric objects as well as area lights and general BSDF models. Although it works well for many scenes, it can exhibit high variance in the presence of particular tricky lighting conditions. For example, consider the setting shown in Figure 15.8: a light source is illuminating a small area on the ceiling such that the rest of the room is only illuminated by indirect lighting bouncing from that area. If we only trace paths starting from the camera, we will almost never happen to sample a path vertex in the illuminated region on the ceiling before we trace a shadow ray to the light. Most of the paths will have no contribution, while a few of them—the ones that happen to hit the small region on the ceiling—will have a large contribution. The resulting image will have high variance.

Difficult lighting settings like this can be handled more effectively by constructing paths that start from the camera on one end and from the light on the other end and are connected in the middle with a visibility ray. This *bidirectional path-tracing* algorithm is a generalization of the standard path-tracing algorithm; for the same amount of computation, it can give substantially lower variance.

The path integral form of the LTE makes it easy to understand how to construct a bidirectional algorithm. As with standard path tracing, the first vertex, p_1, is found by computing the first intersection along the camera ray. The last vertex is found by sampling a point on a light source in the scene. Here, we will label the last vertex as q_1, so that we can construct a path of not initially determined length "backward" from the light.

In the basic bidirectional path-tracing algorithm, we go forward from the camera to create a subpath $p_1, p_2, \ldots, p_i$ and backward from the light to compute a subpath

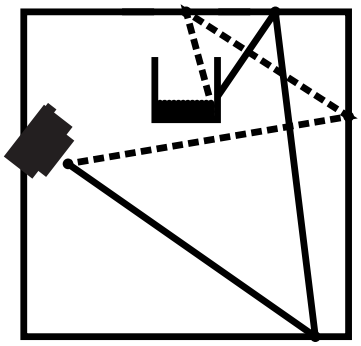


Figure 15.8: A Difficult Case for Path Tracing Starting from the Camera. A light source is illuminating a small area on the ceiling such that only paths with a second-to-last vertex in the area indicated will be able to find illumination from the light. Bidirectional methods, where a path is started from the light and is connected with a path from the camera, can handle situations like these more robustly.

$q_1, q_2, \ldots, q_j$. Each subpath is usually computed incrementally by sampling the BSDF at the previous vertex, although other sampling approaches can be used in the same way as was described for standard path tracing. (Weights for each vertex are computed in the same manner as well.) In either case, in the end, we have a path

$$\bar{p} = p_1, \ldots, p_i, q_j, \ldots, q_1.$$

We need to trace a shadow ray between p_i and q_j to see if they are mutually visible; if so, the path carries light from the light to the camera, and we can evaluate the path's contribution directly.

There are three refinements to this basic algorithm that improve its performance in practice. The first two are analogous to improvements made to path tracing, and the third is a powerful variance reduction technique on top of the improvement from just starting at the light as well.

- First, subpaths can be reused: given a path $p_1, \ldots, p_i, q_j, \ldots, q_1$, transport over all of the paths can be evaluated by connecting all the various combinations of prefixes of the two paths together. If the two paths have i and j vertices, respectively, then a variety of unique paths can be constructed from them, ranging in length from 2 to $i + j$ vertices long. Each such path built this way requires only that a visibility check be performed by tracing a shadow ray between the last vertices of each of the subpaths. (The `BSDF`s for each vertex of both paths must be stored to do this.)
- The second optimization is to ignore the paths generated in the path reuse stage that only use one vertex from the light subpath and instead to use the optimized direct lighting code from the direct lighting integrator. This gives a lower-variance result than using the vertex on the light sampled for the light subpath, since it makes it possible both to use multiple importance sampling with the BSDF and to use well-distributed sampling patterns for this part of the problem.
- The third optimization is to use multiple importance sampling to reweight paths. Recall the example of a light pointed up at the ceiling, indirectly illuminating a room. As described so far, bidirectional path tracing will improve the result substantially by greatly reducing the number of paths with no contribution, since the paths from the light will be effective at finding those light transport routes. However, the image will still suffer from variance due to paths with unexpectedly large contributions—for example, from paths from the camera that happened to find the bright spot in the ceiling. MIS can be applied to solve this, recognizing that for a path with n vertices, there are actually $n - 1$ ways a path with that length could be generated. For example, a four-vertex path could be built from one camera vertex and three light vertices, two of each kind of vertex, or three camera vertices and one light vertex. Given a particular path sampled in a particular way, we can compute the weights for each of the other ways the path could have been generated and apply the balance heuristic.

BSDF 478
MetropolisRenderer 852

`pbrt` doesn't have a bidirectional path-tracing integrator; implementing one is left as an exercise. However, bidirectional path tracing is used in the `MetropolisRenderer` implemented in Section 15.7.

(a) (b) (c)

Figure 15.9: Indoor Scene Rendered with "Instant Global Illumination." (a) Scene rendered with direct illumination only. (b) Using just four virtual light sources with the `IGIIntegrator`. There is indirect lighting, but because so few virtual lights were used, their shadows are distracting. (c) Using 64 virtual lights with the `IGIIntegrator` gives a high-quality result. *(Model courtesy of Florent Boyer.)*

15.4 INSTANT GLOBAL ILLUMINATION

The path-tracing integrator computes unbiased estimates of the scene radiance, though at a cost of requiring from tens to thousands of samples in each pixel in order to compute an image that isn't noisy. This computational cost can be undesirable; this and the following sections will introduce more efficient approaches to solving the LTE, some of them doing so by more effectively reusing intermediate results, sometimes introducing bias.

This section describes the "instant global illumination" (IGI) integrator. The basic idea is to follow a small number of light-carrying paths from the light sources and to construct a number of point light sources at the points where these paths intersect surfaces in the scene. These point sources (also sometimes called *virtual light sources*) are deposited in a way such that their illumination approximates the indirect radiance distribution in the scene. After these lights have been created, when the integrator later needs to compute exitant radiance at a point, it loops over these point light sources and traces shadow rays to see if they are visible, accumulating the indirect illumination that they represent if so. This approach to global illumination has started to see some use in interactive rendering; the "Further Reading" section at the end of this chapter has pointers to relevant papers.

Figure 15.9 shows images of the contemporary house scene rendered with direct lighting and with IGI with different parameter settings. IGI renders indirect lighting effects very efficiently, roughly doubling in computation time for Figure 15.9(c) versus Figure 15.9(a), and without the noise characteristic of algorithms like path tracing or bidirectional path tracing. It can be implemented as an unbiased light transport algorithm; however, because the same virtual lights are used for all of the pixels in the scene, error shows up as systemic error due to correlation as opposed to noise as would be the case for path-tracing algorithms. While this is often an acceptable property for single images, this error can be distracting in animations, where subsequent frames may have noticeably different brightnesses if a different set of lights is generated for each one.

IGIIntegrator 773

In this section, we will show how IGI can be understood as a variant of bidirectional path tracing (see Section 15.3.5). This formulation both makes it easier to understand which cosine terms and such to include in the expressions computed and also makes it easier to see how the approach could be modified. However, while we will use this theory as the foundation for the derivation of the approach, the variable names and such in the implementation will still be in terms of the "virtual light source" approach. The `IGIIntegrator` is implemented in the files `integrators/igi.h` and `integrators/igi.cpp`. As usual, the standard parts of the implementation of the integrator will be elided here, and we will just focus on the parts that are unique to IGI.

The `IGIIntegrator` constructor takes seven parameters. Its implementation is not included here, as it just sets the following member variables from the parameters given.

The first parameter, `nLightPaths`, gives the number of paths to follow from lights to create the virtual lights (the actual number of virtual lights created will generally be larger than this, as each path usually creates multiple lights).

The second parameter, `nLightSets`, describes the number of light sets to create; instead of computing just one set of virtual lights, this integrator actually computes `nLightSets` independent sets of them. To see why doing so is useful, consider an image being rendered at a rate of 16 image samples per pixel: in general, we will expect a better result if each image sample uses a different set of virtual lights, thus incorporating more information about indirect illumination in the scene at little incremental computational cost. Thus, for best results, this value should be the same as the number of samples taken per pixel. [3]

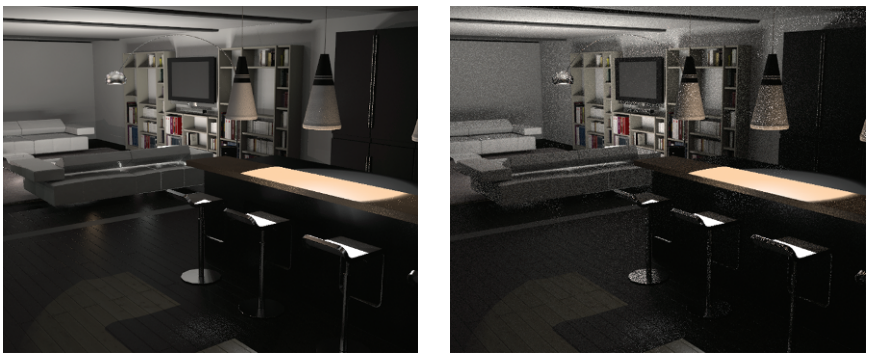
⟨*IGIIntegrator Private Data*⟩ ≡
```
uint32_t nLightPaths, nLightSets;
```

Figure 15.10 shows the example scene rendered again with just one set of lights (a), and with 256 sets (b), both images rendered with 16 samples per pixel. In both cases, image quality suffers. With just one set, all 16 image samples use the same set of virtual lights. As this set only has 10 or so lights in it, indirect lighting cannot be represented accurately, and the virtual lights cast noticeable shadows. With 256 light sets but just 16 pixel samples, as in the second image, the result is noisy, as different pixels will generally use a different set of lights. This case is an interesting illustration of the fact that minimizing numeric error doesn't necessarily minimize the error perceived by a human viewer: increasing numbers of light sets will in general reduce the systemic error due to correlation from re-using the same light set many times. However, this error reduction introduces noise, which is more obviously objectionable to a human observer.

Part of the IGI lighting computation involves evaluating an expression that includes the $G(\mathrm{p} \leftrightarrow \mathrm{p}')$ term that was defined in Equation (15.3). This term can exhibit an *weak singularity* when the distance between p and p′ is small, such that one over the squared distance between them gives a very large number. To avoid bright image artifacts when

IGIIntegrator 773

3 The integrator depends on a well-written `Sampler` to generate a stratified set of samples for the usual case where the number of pixel samples is the same as the number of virtual light sets. Otherwise, the one-to-one relationship between the two may be lost.



(a) (b)

Figure 15.10: Varying the Number of Light Sets with Instant Global Illumination. Scene rendered with 16 samples per pixel. (a) If just one light set is generated, all 16 pixel samples use the same set of virtual lights and aren't able to incorporate as much information about the illumination as if they each used different sets. (b) However, with 256 light sets, the pixel samples in each pixel can use only a subset of them, so that different pixels use different sets for their lighting, introducing noise to the final result. *(Model courtesy of Florent Boyer.)*

this is the case, a different sampling strategy is used whenever the G term is greater than `gLimit`, taking `nGatherSamples` in that case.

⟨*IGIIntegrator Private Data*⟩ +≡
```
float gLimit;
int nGatherSamples;
```

Finally, `rrThreshold` is a Russian roulette threshold used to skip tracing rays to virtual lights that make a small contribution to a particular point where exitant radiance is being computed, and `maxSpecularDepth` is used to terminate specular reflection after a few bounces in the usual manner.

⟨*IGIIntegrator Private Data*⟩ +≡
```
float rrThreshold;
int maxSpecularDepth;
```

In the integrator's `Li()` method, the typical sampling approach is used for the direct lighting computation, where samples are taken from all light sources. In addition to the usual samples needed for that computation, this integrator also needs a well-distributed 1D sample to choose among the multiple sets of virtual lights and may need samples for the gathering sampling algorithm in the weak singularity case; `RequestSamples()` handles notifying the `Sampler` of all of these needed sets of samples.

⟨*IGIIntegrator Method Definitions*⟩ ≡
```
void IGIIntegrator::RequestSamples(Sampler *sampler, Sample *sample,
                                   const Scene *scene) {
    ⟨Allocate and request samples for sampling all lights 743⟩
    vlSetOffset = sample->Add1D(1);
```

BSDFSampleOffsets 706
IGIIntegrator 773
IGIIntegrator::
gatherSampleOffset 775
IGIIntegrator::
nGatherSamples 774
IGIIntegrator::
vlSetOffset 775
Sample 343
Sample::Add1D() 344
Sampler 340
Scene 22

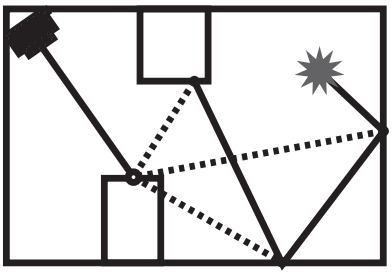


Figure 15.11: Connecting Paths with the `IGIIntegrator`. With the "instant global illumination" algorithm, a number of light-carrying paths are constructed from the light sources; the virtual lights created by such a path are indicated with filled circles here. When a point is being shaded (open circle), shadow rays (dashed lines) are traced between the point and the vertices of the light path to compute indirect illumination. Shadow rays to the light source itself are handled separately, using the direct lighting techniques from Section 15.1.

```
    if (sampler) nGatherSamples = sampler->RoundSize(nGatherSamples);
    gatherSampleOffset = BSDFSampleOffsets(nGatherSamples, sample);
}
```

⟨*IGIIntegrator Private Data*⟩ +≡

```
int vlSetOffset;
BSDFSampleOffsets gatherSampleOffset;
```

15.4.1 CREATING THE VIRTUAL LIGHT SOURCES

The task of the `Preprocess()` routine is to follow light-carrying paths from the light sources and to create the virtual point sources. To understand which values it needs to precompute and store with each virtual light, recall from the discussion of bidirectional path tracing in Section 15.3.5 that we can write the exitant radiance from a point p_1 to a point on the film plane p_0 as an infinite sum over light-carrying paths where each path is an integral over points on the surfaces of objects in the scene. IGI corresponds to the case of bidirectional path tracing where the path from the camera has only one segment and the light path has one or more segments.[4] Figure 15.11 shows an example where the light path has three vertices and has been connected to a camera path of one segment with shadow rays to the vertices of the light path (dotted line). The camera segment and the connecting segment are uniquely determined given the camera ray, its intersection point, and the vertex of the light path it is being connected to.

In this approach, there is only one way to generate any path of length n segments: one camera segment from p_0 to p_1, one connecting segment from p_1 to p_2, and $n - 2$ light segments from p_2 to p_n. Applying the Monte Carlo estimator, the Monte Carlo estimate

BSDFSampleOffsets 706

4 Note that direct lighting and specular reflection thus aren't accounted for here but are easily handled separately with conventional approaches.

of a path's contribution, Equation (15.6), can be written for this case as

$$P(\bar{\mathrm{p}}_n) = \alpha\, f(\mathrm{p}_3 \to \mathrm{p}_2 \to \mathrm{p}_1) G(\mathrm{p}_2 \leftrightarrow \mathrm{p}_1) f(\mathrm{p}_2 \to \mathrm{p}_1 \to \mathrm{p}_0), \tag{15.9}$$

where

$$\alpha = \frac{L_e(\mathrm{p}_n \to \mathrm{p}_{n-1}) f(\mathrm{p}_n \to \mathrm{p}_{n-1} \to \mathrm{p}_{n-2}) |\cos\theta_{n-1}|}{p_A(\mathrm{p}_n)} \times \left(\prod_{i=3}^{n-2} \frac{f(\mathrm{p}_{i+1} \to \mathrm{p}_i \to \mathrm{p}_{i-1})\, |\cos\theta_i|}{p_\omega(\mathrm{p}_{i+1} - \mathrm{p}_i)} \right).$$

The α term is independent of both the segment from the camera to the first visible point p_1 as well as the segment from that point to the last vertex of the light path p_2. Therefore, if we precompute this term, then computing the effect of the corresponding virtual light just requires being able to evaluate the two remaining BSDF terms and the geometric term $G(\mathrm{p}_2 \leftrightarrow \mathrm{p}_1)$.

The `Preprocess()` method creates `nLightSets` sets of paths from the lights in the scene, where each set has `nLightPaths` paths from lights. Each such light path will generally lead to multiple virtual light sources, as one virtual light is created for each vertex in the path after the one on the light source. (Russian roulette is used to terminate paths in an unbiased manner.) Because this step usually requires tracing only a small number of rays, we haven't parallelized it by decomposing it into tasks.

⟨*IGIIntegrator Method Definitions*⟩ +≡

```
void IGIIntegrator::Preprocess(const Scene *scene, const Camera *camera,
                               const Renderer *renderer) {
    if (scene->lights.size() == 0) return;
    MemoryArena arena;
    RNG rng;
    ⟨Compute samples for emitted rays from lights 777⟩
    ⟨Precompute information for light sampling densities 777⟩
    for (uint32_t s = 0; s < nLightSets; ++s) {
        for (uint32_t i = 0; i < nLightPaths; ++i) {
            ⟨Follow path i from light to create virtual lights 777⟩
        }
    }
}
```

To ensure a good sampling of indirect light, it's helpful to use well-distributed samples to choose the initial points on the light sources and their directions. The `LDShuffleScrambled*D()` functions work well to compute samples for generating the initial rays from the lights. Using them in this manner ensures not only that the samples used for each particular set of virtual lights are well distributed but also that in the aggregate over all of the paths the samples are globally well distributed as well.

Camera 302
IGIIntegrator 773
MemoryArena 1015
Renderer 24
RNG 1003
Scene 22

⟨*Compute samples for emitted rays from lights*⟩ ≡ 776
```
vector<float> lightNum(nLightPaths * nLightSets);
vector<float> lightSampPos(2 * nLightPaths * nLightSets, 0.f);
vector<float> lightSampComp(nLightPaths * nLightSets, 0.f);
vector<float> lightSampDir(2 * nLightPaths * nLightSets, 0.f);
LDShuffleScrambled1D(nLightPaths, nLightSets, &lightNum[0], rng);
LDShuffleScrambled2D(nLightPaths, nLightSets, &lightSampPos[0], rng);
LDShuffleScrambled1D(nLightPaths, nLightSets, &lightSampComp[0], rng);
LDShuffleScrambled2D(nLightPaths, nLightSets, &lightSampDir[0], rng);
```

The efficiency of the algorithm is also improved if the probability of starting a path from a light source is related to the amount of power it emits in comparison to the power emitted by the other lights. (The intensity of the corresponding virtual lights must be reduced appropriately to counterbalance the fact that more virtual lights are created for bright light sources.) Increasing the probability of generating paths from bright lights naturally leads to more virtual lights being created for those lights. The `ComputeLightSamplingCDF()` routine computes a discrete probability density for sampling each light according to its power, returning a `Distribution1D` to represent the distribution.

⟨*Precompute information for light sampling densities*⟩ ≡ 776
```
Distribution1D *lightDistribution = ComputeLightSamplingCDF(scene);
```

After a light source is chosen and an initial ray leaving it is sampled, the path is followed, with virtual lights stored at each vertex, until the ray leaves the scene or the path is terminated with Russian roulette.

⟨*Follow path* `i` *from light to create virtual lights*⟩ ≡ 776
```
int sampOffset = s*nLightPaths + i;
⟨Choose light source to trace virtual light path from 777⟩
⟨Sample ray leaving light source for virtual light path 778⟩
Intersection isect;
while (scene->Intersect(ray, &isect) && !alpha.IsBlack()) {
    ⟨Create virtual light and sample new ray for path 778⟩
}
arena.FreeAll();
```

`Distribution1D::SampleDiscrete()` returns a light number to use based on the given sample value and the light CDF computed above.

⟨*Choose light source to trace virtual light path from*⟩ ≡ 777
```
float lightPdf;
int ln = lightDistribution->SampleDiscrete(lightNum[sampOffset],
                                           &lightPdf);
Light *light = scene->lights[ln];
```

Given a particular light, the `Light::Sample_L()` routine gives the ray leaving the light, the radiance it is carrying, and the probability density for sampling the ray. The product of this probability density and the probability density for starting a path from the chosen

ComputeLightSamplingCDF() 709
Distribution1D 648
Distribution1D::SampleDiscrete() 650
Intersection 186
Light 606
MemoryArena::FreeAll() 1017
Scene::Intersect() 23
Scene::lights 23
Spectrum::IsBlack() 265

light, `lightPdf`, gives the overall probability density for sampling the ray from all of the lights.

After the initial ray from the light, the process for incrementally sampling the vertices of the paths is generally similar to the approach used in the `PathIntegrator`. Here, the `alpha` variable tracks the current weight of the path, accounting for radiance emitted by the light source along the initial ray and the product of BSDF terms and sampling densities (i.e., the current value of α in Equation (15.9)).

⟨*Sample ray leaving light source for virtual light path*⟩ ≡ 777
```
RayDifferential ray;
float pdf;
LightSample ls(lightSampPos[2*sampOffset], lightSampPos[2*sampOffset+1],
               lightSampComp[sampOffset]);
Normal Nl;
Spectrum alpha = light->Sample_L(scene, ls, lightSampDir[2*sampOffset],
                                 lightSampDir[2*sampOffset+1],
                                 camera->shutterOpen, &ray, &Nl, &pdf);
if (pdf == 0.f || alpha.IsBlack()) continue;
alpha /= pdf * lightPdf;
```

At each path vertex, the BSDF is found and the ray's weight is updated to account for attenuation through participating media. (Additional radiance due to in-scattering isn't included in the implementation here.)

⟨*Create virtual light and sample new ray for path*⟩ ≡ 777
```
alpha *= renderer->Transmittance(scene, RayDifferential(ray), NULL,
                                 rng, arena);
Vector wo = -ray.d;
BSDF *bsdf = isect.GetBSDF(ray, arena);
⟨Create virtual light at ray intersection point 779⟩
⟨Sample new ray direction and update weight for virtual light path 779⟩
```

Given the intersection at a surface in the scene, a `VirtualLight` object is created to represent the corresponding virtual light source. Given that `alpha` holds the path's contribution up to its arrival at the current point on the surface, we just need to store enough information so that the geometry term and two BSDF terms in that equation can be evaluated given a one-segment path from the camera. Thus, the point p_2 and surface normal at that point must be stored in addition to the `alpha` value, so that the geometric term $G(p_2 \leftrightarrow p_1)$ can be computed.

We'll make a further simplification to avoid needing to be able to evaluate the BSDF at the last vertex of the light path, $f(p_3 \rightarrow p_2 \rightarrow p_1)$. (In a sense, the product of this BSDF and the path's `alpha` vaue at p_2 can be thought of as defining the directionally varying radiant intensity of the virtual point light.) Here, we will represent this term with a constant Lambertian BSDF based on the hemispherical–directional reflectance of the surface, which makes it possible to include the $f(p_3 \rightarrow p_2 \rightarrow p_1)$ BSDF term in

BSDF 478
Intersection::GetBSDF() 484
LightSample 710
Normal 65
PathIntegrator 766
RayDifferential 69
Renderer::Transmittance() 25
Spectrum 263
Vector 57

the partial path contribution stored with the virtual light.[5] As long as the surfaces in the scene aren't too specular, the approximation made here works well.

⟨*Create virtual light at ray intersection point*⟩ ≡ 778
```
Spectrum contrib = alpha * bsdf->rho(wo, rng) / M_PI;
virtualLights[s].push_back(VirtualLight(isect.dg.p, isect.dg.nn, contrib,
                                        isect.rayEpsilon));
```

⟨*IGIIntegrator Private Data*⟩ +≡
```
vector<vector<VirtualLight> > virtualLights;
```

⟨*IGIIntegrator Local Structures*⟩ ≡
```
struct VirtualLight {
    VirtualLight(const Point &pp, const Normal &nn, const Spectrum &c,
                 float reps)
        : p(pp), n(nn), pathContrib(c), rayEpsilon(reps) { }
    Point p;
    Normal n;
    Spectrum pathContrib;
    float rayEpsilon;
};
```

Sampling the outgoing direction leaving an intersection and computing its updated weight are generally similar to the path sampling computation in the `PathIntegrator`. If the path continues, `alpha` is updated to incorporate the next term of the product term in the definition of α after Equation (15.9).

⟨*Sample new ray direction and update weight for virtual light path*⟩ ≡ 778
```
Vector wi;
float pdf;
BSDFSample bsdfSample(rng);
Spectrum fr = bsdf->Sample_f(wo, &wi, bsdfSample, &pdf);
if (fr.IsBlack() || pdf == 0.f)
    break;
Spectrum contribScale = fr * AbsDot(wi, bsdf->dgShading.nn) / pdf;
⟨Possibly terminate virtual light path with Russian roulette 780⟩
alpha *= contribScale / rrProb;
ray = RayDifferential(isect.dg.p, wi, ray, isect.rayEpsilon);
```

AbsDot() 61
BSDF::dgShading 479
BSDF::rho() 482
BSDF::Sample_f() 706
BSDFSample 705
DifferentialGeometry::nn 102
DifferentialGeometry::p 102
IGIIntegrator 773
Intersection::dg 186
Intersection::rayEpsilon 186
MemoryArena 1015
M_PI 1002
Normal 65
Point 63
RayDifferential 69
Spectrum 263
Spectrum::IsBlack() 265
Vector 57
VirtualLight 779

One small difference here compared to other integrators in this chapter is how Russian roulette is used for path termination. In the others, the termination probability in the Russian roulette test is generally constant after a fixed number of bounces have occurred. Here, the termination probability is one minus the luminance of the product of the BSDF's value and the cosine term divided by the sampling PDF for the outgoing direction. Thus, if the surface has a high albedo—it reflects most of the incident illumination—the probability of continuing the path is high, and if the albedo is small

5 The alternative would be to retain the BSDFs, which would require not freeing the space allocated by the `MemoryArena` during preprocessing here; the arena would instead likely need to be a member variable of the `IGIIntegrator`.

it is likely that the path will be terminated. This is an intuitively efficient property; if the path is carrying a large amount of light, it should be likely to continue.

An interesting implication of this approach is that the luminance of the path contribution value remains constant after each surface intersection. For example, consider a surface where the luminance of `contribScale` is 0.1. The path will be terminated with 90% probability. For the 10% where the path continues, `alpha` will be scaled by `contribScale`, to account for the light reflection, and divided by 0.1, to account for Russian roulette. Because the luminance of `contribScale` is 0.1, the net result will be that the luminance of the path is unchanged.

Thus, surfaces that reflect little light lead to fewer paths leaving them, rather than paths with a lower contribution, and all of the virtual lights created will have similar path contributions and will thus tend to make an equal contribution to the final exitant radiance values computed. This generally leads to better results than if some of them were thousands of times brighter than others, for example.

⟨*Possibly terminate virtual light path with Russian roulette*⟩ ≡ 779
```
float rrProb = min(1.f, contribScale.y());
if (rng.RandomFloat() > rrProb)
    break;
```

15.4.2 RENDERING WITH VIRTUAL LIGHT SOURCES

Most of the `IGIIntegrator::Li()` implementation isn't included here; it samples direct lighting in the usual manner, recursively tracing rays to account for perfect specular reflection and refraction, as integrators like `DirectLightingIntegrator` do. After the direct lighting contribution has been computed at the point being shaded, the contribution from the virtual light sources is found by the ⟨*Compute indirect illumination with virtual lights*⟩ fragment. This fragment starts by using the appropriate sample value from `sample` to determine which of the sets of virtual lights to use for this point.

⟨*Compute indirect illumination with virtual lights*⟩ ≡
```
uint32_t lSet = min(uint32_t(sample->oneD[vlSetOffset][0] * nLightSets),
                    nLightSets-1);
for (uint32_t i = 0; i < virtualLights[lSet].size(); ++i) {
    const VirtualLight &vl = virtualLights[lSet][i];
    ⟨Compute virtual light's tentative contribution Llight 782⟩
    ⟨Possibly skip virtual light shadow ray with Russian roulette 782⟩
    ⟨Add contribution from VirtualLight vl 782⟩
}
if (ray.depth < maxSpecularDepth) {
    ⟨Do bias compensation for bounding geometry term 783⟩
}
```

Since the BSDF of the last vertex of the light path was approximated by a constant Lambertian term, there are two remaining factors to evaluate from Equation (15.9) in order to compute a light path's contribution to the current camera path:

$$f(p_0 \to p_1 \to p_2) G(p_1 \leftrightarrow p_2).$$

RNG::RandomFloat() 1003
Spectrum::y() 273
VirtualLight 779

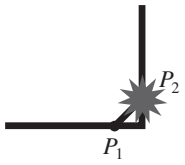


Figure 15.12: The Weak Singularity from the $G(p_1 \leftrightarrow p_2)$ Term. When the virtual light at p_2 and the point being shaded at p_1 are close together, this term can become very large due to the $1/r^2$ term, leading to bright splotches in rendered images. The `IGIIntegrator` clamps G to `gLimit` to avoid this problem, using a different sampling strategy that handles this case more robustly.

(a) (b)

Figure 15.13: Effect of the Weak Singularity When Evaluating Virtual Light Contributions. (a) If a virtual light is close to the point being shaded, the G term may be large, leading to bright splotches in the image. (b) Clamping G to be no larger than a fixed value eliminates these splotches but makes the rendering algorithm biased. Note, for example, that the top of the cushion in the upper right has become dimmer.

These are easily computed with the values available to the integrator and in the `VirtualLight` structure.

Here, we can see where the `gLimit` parameter is used by the implementation: if the points p_1 and p_2 are close to each other, as illustrated in Figure 15.12, G may become very large, leading to bright splotches in the image, as shown by Figure 15.13(a). For now, G is clamped so that it is no larger than `gLimit`, which eliminates these artifacts. However, this clamping makes the algorithm biased; images will be slightly darker on average than they should be (Figure 15.13(b)). The fragment ⟨*Do bias compensation for bounding geometry term*⟩, to be introduced shortly, accounts for this difference, making the final result again unbiased.

⟨*Compute virtual light's tentative contribution* `Llight`⟩ ≡ 780
```
float d2 = DistanceSquared(p, vl.p);
Vector wi = Normalize(vl.p - p);
float G = AbsDot(wi, n) * AbsDot(wi, vl.n) / d2;
G = min(G, gLimit);
Spectrum f = bsdf->f(wo, wi);
if (G == 0.f || f.IsBlack()) continue;
Spectrum Llight = f * G * vl.pathContrib / virtualLights[lSet].size();
RayDifferential connectRay(p, wi, ray, isect.rayEpsilon,
                           sqrtf(d2) * (1.f - vl.rayEpsilon));
Llight *= renderer->Transmittance(scene, connectRay, NULL, rng, arena);
```

If the light's contribution to outgoing radiance at the point is low, Russian roulette is used to sometimes avoid tracing the shadow ray for it. Thus, less time is spent tracing shadow rays for unimportant lights.

⟨*Possibly skip virtual light shadow ray with Russian roulette*⟩ ≡ 780
```
if (Llight.y() < rrThreshold) {
    float continueProbability = .1f;
    if (rng.RandomFloat() > continueProbability)
        continue;
    Llight /= continueProbability;
}
```

Finally, the visibility term is checked and the virtual light's contribution is added if it is not occluded.

⟨*Add contribution from* `VirtualLight vl`⟩ ≡ 780
```
if (!scene->IntersectP(connectRay))
    L += Llight;
```

To understand how to compensate for clamping the value of G above, consider the general setting of computing an area integral of some function f scaled by a geometric term,

$$\int_A f(\mathrm{p}, \mathrm{p}')\, G(\mathrm{p} \leftrightarrow \mathrm{p}')\, \mathrm{d}A. \tag{15.10}$$

When performing Monte Carlo integration, we'd like to clamp G when it becomes too large in order to avoid artifacts when a sample p' happens to be very close to p. To do so in a way that doesn't introduce bias, we can first use the identity $a = \min(a, b) + \max(a - b, 0)$ to rewrite Equation (15.10) as

$$\int_A f(\mathrm{p}, \mathrm{p}') \min(G(\mathrm{p}_\mathrm{i} \leftrightarrow \mathrm{p}_1), G_\mathrm{limit})\, \mathrm{d}A + \int_A f(\mathrm{p}, \mathrm{p}') \max(G(\mathrm{p}_\mathrm{i} \leftrightarrow \mathrm{p}_1) - G_\mathrm{limit}, 0)\, \mathrm{d}A. \tag{15.11}$$

The first term doesn't exhibit the weak singularity from G. We can then rewrite the second term as an integral over directions rather than area, using the Jacobian for the relationship between surface area and solid angle, $r^2/|\cos\theta|$,

AbsDot() 61
BSDF::f() 481
DistanceSquared() 65
IGIIntegrator::gLimit 774
IGIIntegrator::rrThreshold 774
IGIIntegrator::virtualLights 779
Intersection::rayEpsilon 186
RayDifferential 69
Renderer::Transmittance() 25
RNG::RandomFloat() 1003
Scene::IntersectP() 24
Spectrum 263
Spectrum::IsBlack() 265
Spectrum::y() 273
Vector 57
Vector::Normalize() 63
VirtualLight::pathContrib 779
VirtualLight::rayEpsilon 779

$$\int_{S^2} f(\mathrm{p}, \mathrm{p}') \max(G(\mathrm{p} \leftrightarrow \mathrm{p}') - G_{\mathrm{limit}}, 0) \frac{\| \mathrm{p} - \mathrm{p}' \|^2}{|\cos\theta|} \, \mathrm{d}\omega'$$

where $\mathrm{p}' = t(\mathrm{p}, \omega')$, using the trace function t from Section 15.2.1. This expression can be rewritten using the definition of G as

$$\int_{S^2} f(\mathrm{p}, \mathrm{p}') \frac{\max(G(\mathrm{p} \leftrightarrow \mathrm{p}') - G_{\mathrm{limit}}, 0)}{G(\mathrm{p} \leftrightarrow \mathrm{p}')} |\cos\theta'| \, \mathrm{d}\omega', \tag{15.12}$$

where θ' is the angle between the ray from p to p′ with the surface normal at p′. Note that this form doesn't exhibit the weak singularity as p′ approaches p.

Now we can apply the approach embodied by Equation (15.12) to the problem at hand. When we use the virtual light sources to evaluate Equation (15.9) to compute indirect illumination at a point, we're evaluating an estimate of the path contribution function, Equation (15.6). It just so happens that we're using the same sets of samples $\mathrm{p}_n, \ldots, \mathrm{p}_2$ for all points being shaded. By clamping G in this computation, we have evaluated the first term of Equation (15.11), with f defined appropriately. Thus, we now need to add in the effect of Equation (15.12); this task is handled by ⟨*Do bias compensation for bounding geometry term*⟩, using the BSDF's sampling method to choose directions.

⟨*Do bias compensation for bounding geometry term*⟩ ≡ 780
```
int nSamples = (ray.depth == 0) ? nGatherSamples : 1;
for (int i = 0; i < nSamples; ++i) {
    Vector wi;
    float pdf;
    BSDFSample bsdfSample = (ray.depth == 0) ?
        BSDFSample(sample, gatherSampleOffset, i) : BSDFSample(rng);
    Spectrum f = bsdf->Sample_f(wo, &wi, bsdfSample,
                                &pdf, BxDFType(BSDF_ALL & ~BSDF_SPECULAR));
    if (!f.IsBlack() && pdf > 0.f) {
        ⟨Trace ray for bias compensation gather sample 783⟩
        ⟨Add bias compensation ray contribution to radiance sum 784⟩
    }
}
```

These rays are relatively inexpensive to trace since their maximum length can be bounded; given a particular value of `gLimit`, we can compute the distance beyond which G will always be less than `gLimit`, such that the value of the second term of Equation (15.11) will be zero. The ray's maximum distance is set to this value; thus, many of the rays often won't intersect any geometry, so no shading calculation needs to be performed for them.

⟨*Trace ray for bias compensation gather sample*⟩ ≡ 783
```
float maxDist = sqrtf(AbsDot(wi, n) / gLimit);
RayDifferential gatherRay(p, wi, ray, isect.rayEpsilon, maxDist);
Intersection gatherIsect;
Spectrum Li = renderer->Li(scene, gatherRay, sample, rng, arena,
                           &gatherIsect);
if (Li.IsBlack()) continue;
```

AbsDot() 61
BSDF::Sample_f() 706
BSDFSample 705
BSDF_ALL 428
BSDF_SPECULAR 428
BxDFType 428
IGIIntegrator::
gatherSampleOffset 775
IGIIntegrator::
nGatherSamples 774
Intersection 186
Intersection::rayEpsilon 186
Ray::depth 67
RayDifferential 69
Renderer::Li() 25
Spectrum 263
Spectrum::IsBlack() 265
Vector 57

⟨*Add bias compensation ray contribution to radiance sum*⟩ ≡ 783

```
float Ggather = AbsDot(wi, n) * AbsDot(-wi, gatherIsect.dg.nn) /
    DistanceSquared(p, gatherIsect.dg.p);
if (Ggather - gLimit > 0.f && !isinf(Ggather)) {
    float gs  = (Ggather - gLimit) / Ggather *
                AbsDot(-wi, gatherIsect.dg.nn);
    L += f * Li * (AbsDot(wi, n) * gs / (nSamples * pdf));
}
```

15.5 IRRADIANCE CACHING

With unbiased light transport algorithms like path tracing, some scenes can take a large number of rays (and corresponding compute time) to generate images without objectionable noise. One approach to this problem has been the development of biased approaches to solving the LTE. These approaches generally reuse previously computed results over multiple exitant radiance computations, even when the values used don't give the precise quantity that needs to be computed (for example, by reusing an illumination value from a nearby point under the assumption that illumination is slowly changing). Irradiance caching, described in this section, and photon mapping, described in the next, have been two successful biased methods for light transport.

By introducing bias, these methods produce images without the high-frequency noise artifacts that unbiased Monte Carlo techniques are prone to. They can often create good-looking images using relatively little additional computation compared to basic techniques like Whitted ray tracing. This efficiency comes at a price, however: one key characteristic of unbiased Monte Carlo techniques is that variance decreases in a predictable and well-characterized manner as more samples are taken. As such, if an image was computed with an unbiased technique and has no noise, we can be extremely confident that the image correctly represents the lighting in the scene. With a biased solution method, however, error estimates aren't well defined for the approaches that have been developed so far; if the image doesn't have visual artifacts, it still may have error. Even worse, given an image with artifacts, increasing the sampling rate with a biased technique doesn't necessarily eliminate artifacts in a predictable way.

The *irradiance caching* algorithm is based on the observation that, while direct lighting often changes rapidly from point to point (e.g., consider a hard shadow edge), indirect lighting is often much more slowly changing. Therefore, if we compute an accurate representation of indirect light at a sparse set of sample points in the scene and then interpolate nearby samples to compute an approximate representation of indirect light at any particular point being shaded, we can expect that the error introduced by not recomputing indirect lighting everywhere shouldn't be too bad, and we can achieve a substantial computational savings by not recomputing this information at every point. Figure 15.14, which shows the indirect lighting component of a view of the San Miguel scene, illustrates the smoothness of indirect lighting in this environment.

DistanceSquared() 65
IGIIntegrator::gLimit 774

There are two issues that must be addressed in the design of an algorithm such as this one:

Figure 15.14: Indirect Illumination Component of a View of the San Miguel Scene. Note its smoothly varying and slowly changing nature. *(Model courtesy of Guillermo M. Leal Llaguno.)*

1. How is the indirect lighting distribution represented and stored after being computed at a point?
2. When are new representations of indirect light computed, and how often are already existing ones interpolated?

In the irradiance caching algorithm implemented here, indirect lighting is computed at a subset of the points in the scene that are shaded (as opposed to a predetermined set of points) and stored in a spatial data structure. When exitant radiance at a point is being computed, the cache is first searched for one or more acceptable nearby samples, using a set of error metrics to determine if the already existing samples are acceptable. If not enough acceptable samples are found, a new one is computed and added to the data structure.

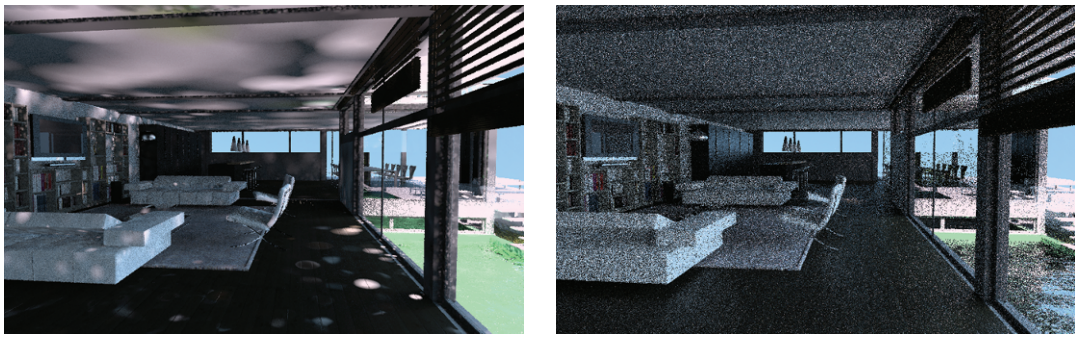
In order to have a compact representation of indirect light, this algorithm only stores the irradiance and the average direction of incident irradiance at each point, rather than a directionally varying radiance distribution, thus reducing the representation of incident illumination to just a single `Spectrum` and a single `Vector`.

To understand the main idea of the approach, first recall that irradiance arriving at one side of a surface with normal $\mathbf{n}$ is

$$E(\mathrm{p}, \mathbf{n}) = \int_{\mathcal{H}^2(\mathbf{n})} L_\mathrm{i}(\mathrm{p}, \omega_\mathrm{i}) \, |\cos \theta_\mathrm{i}| \, \mathrm{d}\omega_\mathrm{i}.$$

It is in a sense a weighted average of incoming radiance at a point, giving a sense of the aggregate illumination. Now consider the reflection component of the scattering equation for a reflective surface:

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



(a) (b)

Figure 15.15: Contemporary house interior scene rendered with (a) irradiance caching and (b) path tracing, both with approximately the same amount of computation time. We have intentionally limited the time spent rendering these in order to compare the image artifacts of the two methods when they are not able to take enough samples to sufficiently sample the scene radiance distribution. Path tracing has high-frequency noise, while irradiance caching suffers from circular blotches. *(Model courtesy of Florent Boyer.)*

If the surface is Lambertian, the BSDF is constant, and we have

$$\begin{aligned} L_o(\mathrm{p}, \omega_o) &= c \int_{\mathcal{H}^2(\mathbf{n})} L_i(\mathrm{p}, \omega_i) \, |\cos \theta_i| \, d\omega_i \\ &= c \, E(\mathrm{p}, \mathbf{n}). \end{aligned} \qquad [15.13]$$

In other words, for perfectly diffuse materials, the irradiance alone is enough information to exactly compute the reflection from the surface due to a particular incident lighting distribution. Thus, interpolating irradiance values can compute reflection from perfectly diffuse surfaces, with the only source of error being the error from interpolation. In the implementation below, the average incident lighting direction will be used to make the approach work well for mostly diffuse or slightly glossy surfaces; the details will be explained shortly.

Figure 15.15 shows the irradiance caching integrator described in this section in action. Figure 15.15(a) was rendered using the irradiance cache, and Figure 15.15(b) with path tracing, both using approximately the same amount of computation. The artifacts from undersampling with the irradiance cache are quite different than the variance from path tracing. Figure 15.16 shows the locations at which irradiance estimates were computed for this example.

⟨*IrradianceCacheIntegrator Declarations*⟩ ≡

```
class IrradianceCacheIntegrator : public SurfaceIntegrator {
public:
    ⟨IrradianceCacheIntegrator Public Methods⟩
private:
    ⟨IrradianceCacheIntegrator Private Data 787⟩
    ⟨IrradianceCacheIntegrator Private Methods⟩
};
```

IrradianceCacheIntegrator 786
SurfaceIntegrator 740

The `IrradianceCacheIntegrator` constructor initializes the member variables below with values passed into it; therefore, it is elided here. The sample pixel spacing member

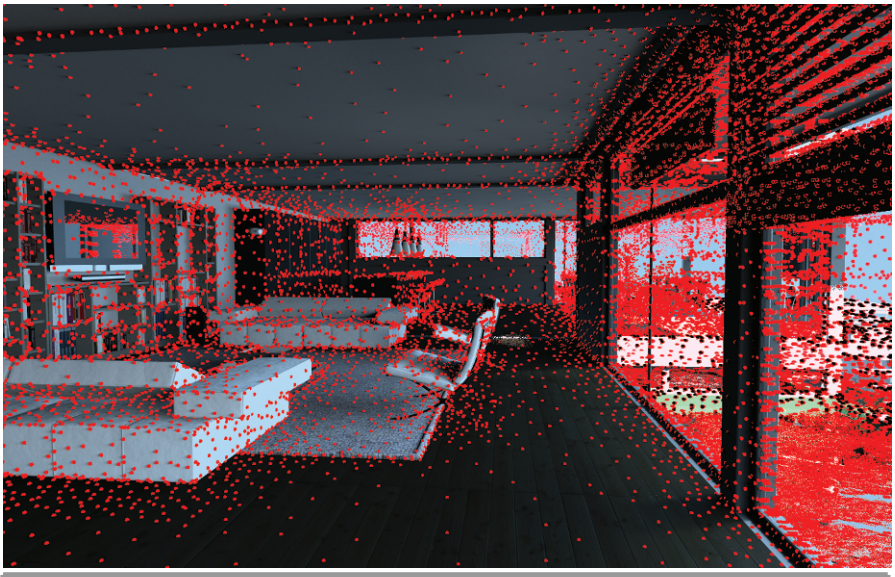


Figure 15.16: The Positions at Which Irradiance Estimates Were Computed for the Image in Figure 15.15. Note that they are mostly near the corners, where indirect illumination is changing most rapidly.

variables control the spacing of irradiance samples, measured in pixels on the image: they ensure that no two samples are closer than a minimum spacing as seen from the camera, but that no two are farther than a maximum. In general, extra samples less than a pixel or so apart have little visual impact on the image and aren't worth the computational expense, and not having enough samples at a minimum pixel frequency risks artifacts from excessive interpolation.

The minimum weight and `cosMaxSampleAngleDifference` values control the error metric used in determining if a particular irradiance sample can be used at a given point without introducing excessive error. Their use will be explained in the code that uses them. The number of Monte Carlo samples used to compute the irradiance estimates is controlled by `nSamples`, and the maximum depth parameters control the ray tree depths for specular reflections and the indirect lighting calculation done to compute irradiance samples. Finally, the class stores a reader–writer mutex to protect the octree storing irradiance samples from concurrent updates by multiple threads.

⟨*IrradianceCacheIntegrator Private Data*⟩ ≡ 786

```
float minSamplePixelSpacing, maxSamplePixelSpacing;
float minWeight, cosMaxSampleAngleDifference;
int nSamples, maxSpecularDepth, maxIndirectDepth;
mutable RWMutex *mutex;
```

DirectLightingIntegrator 742
RWMutex 1039

This integrator reuses the direct lighting routines that were defined in the direct lighting integrator, although here all lights are always sampled. Code fragments to request samples for this computation are reused from the `DirectLightingIntegrator` as well.

⟨*IrradianceCacheIntegrator Method Definitions*⟩ ≡
```
void IrradianceCacheIntegrator::RequestSamples(Sampler *sampler,
        Sample *sample, const Scene *scene) {
    ⟨Allocate and request samples for sampling all lights 743⟩
}
```

⟨*IrradianceCacheIntegrator Private Data*⟩ +≡ 786
```
⟨Declare sample parameters for light source sampling 743⟩
```

The `Preprocess()` method allocates the octree that stores the irradiance samples. (The octree can't be allocated until the scene has been fully specified and its bounds are known—this isn't the case when the `IrradianceCacheIntegrator`'s constructor runs.) The method expands the bounds by a small amount in each direction so that the octree can gracefully deal with the fact that some of the irradiance samples and some of the lookup points may be marginally outside the scene bounds due to floating-point error from ray intersection computations.

After the octree is allocated, the ⟨*Prime irradiance cache*⟩ fragment executes. This fragment's job is to prefill the irradiance cache with irradiance samples that are likely to be needed when actually rendering the image. If the cache wasn't prefilled but was purely filled on demand, then we'd face the problem that often after a new irradiance sample was computed and added to the cache we would have liked to have had that sample available when computing outgoing radiance from previous points: two immediately adjacent image samples that should actually have the same incident irradiance used in shading them may instead use very different values—the first sample interpolating among existing irradiance values, just passing the error threshold, and the second not passing the error threshold for interpolation and thus computing an entirely new sample.

This fragment creates a `Sampler` that takes a single sample per pixel, decomposes the image plane into tiles, and launches a task for each tile. The tasks use the sampler to generate camera rays, find their intersections with objects in the scene, and call the `IrradianceCacheIntegrator::Li()` method. This fragment isn't included here because it is relatively long and similar in form to the `SamplerRenderer` and `SamplerRendererTasks` that it creates; the main difference is that the value returned from `Li()` is discarded; the method is called purely to cause cache samples to be added.

⟨*IrradianceCacheIntegrator Method Definitions*⟩ +≡
```
void IrradianceCacheIntegrator::Preprocess(const Scene *scene,
        const Camera *camera, const Renderer *renderer) {
    BBox wb = scene->WorldBound();
    Vector delta = .01f * (wb.pMax - wb.pMin);
    wb.pMin -= delta;
    wb.pMax += delta;
    octree = new Octree<IrradianceSample *>(wb);
    ⟨Prime irradiance cache⟩
}
```

⟨*IrradianceCacheIntegrator Private Data*⟩ +≡ 786
```
mutable Octree<IrradianceSample *> *octree;
```

BBox 70
BBox::pMax 71
BBox::pMin 71
Camera 302
IrradianceCacheIntegrator 786
IrradianceCacheIntegrator::Li() 789
IrradianceCacheIntegrator::octree 788
IrradianceSample 793
Octree 1023
Renderer 24
Sample 343
Sampler 340
SamplerRenderer 25
SamplerRendererTask 29
Scene 22
Scene::WorldBound() 24
Vector 57

15.5.1 RENDERING WITH THE IRRADIANCE CACHE

We won't include most of the irradiance cache's `Li()` method here—it does the usual BSDF evaluation and so forth—but will just focus on its key two fragments, ⟨*Compute direct lighting for irradiance cache*⟩ and ⟨*Compute indirect lighting for irradiance cache*⟩. For direct lighting, the `UniformSampleAllLights()` function from Section 15.1 is used to apply multiple importance sampling for the direct lighting estimate.

⟨*Compute direct lighting for irradiance cache*⟩ ≡
```
L += UniformSampleAllLights(scene, renderer, arena, p, n, wo,
        isect.rayEpsilon, ray.time, bsdf, sample, rng,
        lightSampleOffsets, bsdfSampleOffsets);
```

The `IrradianceCacheIntegrator` partitions the BSDF for the indirect lighting computation. Perfect specular reflection is handled by sampling the BSDF and recursively calling the integrator, just as the `WhittedIntegrator` does. The implementation here uses irradiance caching for both the diffuse and glossy components of the BSDF, thus introducing additional error for the glossy components. As long as the glossiness isn't too close to perfect specular reflection, the error introduced from this approximation is generally acceptable.

⟨*Compute indirect lighting for irradiance cache*⟩ ≡
```
if (ray.depth + 1 < maxSpecularDepth) {
    Vector wi;
    ⟨Trace rays for specular reflection and refraction 46⟩
}
⟨Estimate indirect lighting with irradiance cache 789⟩
```

If the surface has both reflective and transmissive nonspecular components, then reflection and transmission must be handled separately with two independent irradiance values, since the irradiance values from the hemisphere that is needed for reflective surfaces and the hemisphere on the opposite side of the surface that is needed for transmissive surfaces can be completely different. The `indirectLo()` method handles either case, interpolating a value using the cache or computing a new value and using it to compute exitant radiance due to incident irradiance.

Before calling this method, the integrator reorients the normal so that it points in the same hemisphere as the ω_o vector. `indirectLo()` depends on this convention when it generates sample rays over the hemisphere, since it will ensure that all of these rays are in the same hemisphere as `n`.

⟨*Estimate indirect lighting with irradiance cache*⟩ ≡ 789
```
Normal ng = isect.dg.nn;
ng = Faceforward(ng, wo);
⟨Compute pixel spacing in world space at intersection point 790⟩
BxDFType flags = BxDFType(BSDF_REFLECTION | BSDF_DIFFUSE | BSDF_GLOSSY);
L += indirectLo(p, ng, pixelSpacing, wo, isect.rayEpsilon,
                bsdf, flags, rng, scene, renderer, arena);
flags = BxDFType(BSDF_TRANSMISSION | BSDF_DIFFUSE | BSDF_GLOSSY);
L += indirectLo(p, -ng, pixelSpacing, wo, isect.rayEpsilon,
                bsdf, flags, rng, scene, renderer, arena);
```

BSDF_DIFFUSE 428
BSDF_GLOSSY 428
BSDF_REFLECTION 428
BSDF_TRANSMISSION 428
BxDFType 428
DifferentialGeometry::nn 102
Faceforward() 66
Intersection::dg 186
IrradianceCacheIntegrator 786
IrradianceCacheIntegrator::
indirectLo() 790
IrradianceCacheIntegrator::
maxSpecularDepth 787
Normal 65
Ray::depth 67
UniformSampleAllLights() 745
Vector 57

Part of the error metric used in deciding which irradiance samples can be acceptably interpolated is based on the distance between a candidate sample and the current lookup point, expressed in pixels. Recall from Section 2.2.3 that the cross product of two vectors gives the area of the parallelogram that they form; taking the cross product of the $\partial p/\partial x$ and $\partial p/\partial y$ vectors from the `DifferentialGeometry` structure gives the world-space area of a pixel at the point being shaded. If we make the approximation that this region is square, then its square root gives the length of a side—the distance that we need to compute.

⟨*Compute pixel spacing in world space at intersection point*⟩ ≡ 789
```
float pixelSpacing = sqrtf(Cross(isect.dg.dpdx, isect.dg.dpdy).Length());
```

The `indirectLo()` method returns the reflected radiance for the BSDF components specified by the `flags` parameter, computed by finding the irradiance at the given point in the hemisphere by the given normal. A surprising number of additional parameters to the method are needed; many of them are needed for cases where a new irradiance value needs to be computed and added to the cache.

⟨*IrradianceCacheIntegrator Method Definitions*⟩ +≡
```
Spectrum IrradianceCacheIntegrator::indirectLo(const Point &p,
        const Normal &ng, float pixelSpacing, const Vector &wo,
        float rayEpsilon, BSDF *bsdf, BxDFType flags, RNG &rng,
        const Scene *scene, const Renderer *renderer,
        MemoryArena &arena) const {
    if (bsdf->NumComponents(flags) == 0)
        return Spectrum(0.);
    Spectrum E;
    Vector wi;
    ⟨Get irradiance E and average incident direction wi at point p 791⟩
    ⟨Compute reflected radiance due to irradiance and BSDF 791⟩
}
```

The ⟨*Get irradiance* `E` *and average incident direction* `wi` *at point* `p`⟩ fragment will be defined shortly, in the following subsection. Before going into the details of irradiance sample computation, caching, and interpolation, we'll first show how the computed irradiance estimates are used for shading.

Following the approach introduced by Tabellion and Lamorlette (2004), we can approximate the incident irradiance by incident radiance along a single direction—in some sense converting it to an equivalent directional light source. Doing so maintains a very rough approximation of the directional distribution of incident radiance at the point being shaded. The end result for perfectly diffuse surfaces is the same as if we just multiplied the irradiance by the reflectance, but for moderately glossy surfaces this approximation produces better results than assuming a uniform incident distribution of illumination.

Given an irradiance value E and an incident direction ω_{avg}, we can compute the amount of radiance L that must arrive along that direction to give the same irradiance:

$$E(\mathrm{p}, \mathbf{n}) = \int_{\mathcal{H}^2(\mathbf{n})} L(\mathrm{p}, \omega_{\mathrm{i}})\, \delta(\omega_{\mathrm{i}} - \omega_{\text{avg}})\, |\cos\theta_i|\, \mathrm{d}\omega_{\mathrm{i}} = L_{\text{avg}}\, |\cos\theta_{\text{avg}}|,$$

BSDF 478
BSDF::NumComponents() 479
BxDFType 428
Cross() 62
DifferentialGeometry::dpdx 505
DifferentialGeometry::dpdy 505
Intersection::dg 186
IrradianceCacheIntegrator 786
MemoryArena 1015
Normal 65
Point 63
Renderer 24
RNG 1003
Scene 22
Spectrum 263
Vector 57
Vector::Length() 62

so $L_{\text{avg}} = E/|\cos\theta_{\text{avg}}|$. Substituting into the scattering equation, we have

$$\begin{aligned} L_{\text{o}}(\text{p}, \omega_{\text{o}}) &= \int_{\mathcal{H}^2(\mathbf{n})} f(\text{p}, \omega_{\text{o}}, \omega_{\text{i}})\, \delta(\omega_{\text{i}} - \omega_{\text{avg}})\, \frac{E}{|\cos\theta_{\text{avg}}|}\, |\cos\theta_{\text{i}}|\, \text{d}\omega_{\text{i}} \\ &= f(\text{p}, \omega_{\text{o}}, \omega_{\text{avg}}) E(\text{p}, \mathbf{n}). \end{aligned}$$

Thus, computing the reflected radiance is straightforward. The only complication is that the length of the weighted average direction of incident radiance function may be zero; this happens when there is no irradiance at the point. In this case, a black spectrum is returned immediately to avoid calling `Vector::Normalize()` on a zero-length vector, which would in turn return not-a-number values.

⟨*Compute reflected radiance due to irradiance and BSDF*⟩ ≡ 790
```
if (wi.LengthSquared() == 0.f) return Spectrum(0.);
return bsdf->f(wo, Normalize(wi), flags) * E;
```

15.5.2 LOOKUP AND INTERPOLATITON

The `interpolateE()` method attempts to interpolate the irradiance values already in the octree to approximate the irradiance at the given point in the hemisphere with the given normal. If suitable irradiance samples aren't found in the cache, a new sample is computed and added. In either case, `E` is initialized with an approximation of irradiance for this point, `wi` is initialized to the average direction of incident radiance, and exitant radiance can be computed with the approximation developed above. The fragments that define the computation to compute new irradiance samples are defined in Section 15.5.3; this section will focus on the lookup and interpolation process.

⟨*Get irradiance* `E` *and average incident direction* `wi` *at point* `p`⟩ ≡ 790
```
if (!interpolateE(scene, p, ng, &E, &wi)) {
    ⟨Compute irradiance at current point 795⟩
    ⟨Add computed irradiance value to cache 796⟩
}
```

Recall from the `Preprocess()` method that `IrradianceCacheIntegrator` uses an octree data structure to store the estimates. Doing so allows it to efficiently search for all of the already computed irradiance estimates around a point in the scene. The `Octree` template class, which is described in Section A.7 in Appendix A, recursively splits a given bounding box into subregions, refining the current region into eight subregions at each level of the tree by dividing the box in half at the midpoint of its extent along the x, y, and z axes. Each irradiance estimate has an axis-aligned bounding box associated with it, giving the overall area for which it is potentially a valid sample. The octree uses the extent of this box as a guide for an appropriate level of the tree at which to store the sample. Later, given a point to look up nearby irradiance samples for, the octree just needs to traverse the nodes that the point is inside (there is one such node at each level of the tree) and provide the samples overlapping those nodes to be considered for interpolation (Figure 15.17).

Much of the work in the `interpolateE()` method is done by the `Octree::Lookup()` method, which traverses the nodes of the octree that the given point is inside and calls the `operator()` method of the `IrradProcess` object for each `IrradianceSample` in each

BSDF::f() 481
IrradianceCacheIntegrator 786
IrradianceCacheIntegrator::interpolateE() 792
Spectrum 263
Vector::LengthSquared() 62
Vector::Normalize() 63

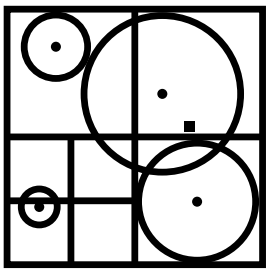


Figure 15.17: Example of Irradiance Sample Storage in 2D (with a Quadtree Rather than an Octree). Each irradiance sample, denoted by a dot, has a maximum distance over which it potentially can contribute irradiance, denoted here by a circle. Samples are stored in the tree nodes that they overlap, and the tree is refined adaptively so that each sample is stored in a small number of nodes. Given a point at which we want to look up nearby irradiance estimates, here shown with a black square, we just need to traverse the tree nodes that the point overlaps, considering all of the irradiance samples stored in these nodes.

of these nodes. `IrradProcess` decides if each sample is acceptable and accumulates the value of the interpolated result. Note that a reader lock is acquired before the lookup is performed; doing so ensures that another thread can't insert new values into the octree as it is being traversed in this thread.

⟨*IrradianceCacheIntegrator Method Definitions*⟩ +≡

```
bool IrradianceCacheIntegrator::interpolateE(const Scene *scene,
        const Point &p, const Normal &n, Spectrum *E,
        Vector *wi) const {
    if (!octree) return false;
    IrradProcess proc(p, n, minWeight, cosMaxSampleAngleDifference);
    RWMutexLock lock(*mutex, READ);
    octree->Lookup(p, proc);
    if (!proc.Successful()) return false;
    *E = proc.GetIrradiance();
    *wi = proc.GetAverageDirection();
    return true;
}
```

The `IrradProcess` structure stores the position and normal of the point being shaded as well as the values of the error parameters. It accumulates information about the interpolated value as it processes candidate irradiance samples. Its constructor is straightforward and not included here; it initializes the member variables with the parameter values given or sets them to initial values.

⟨*IrradianceCacheIntegrator Local Declarations*⟩ ≡

```
struct IrradProcess {
    ⟨IrradProcess Public Methods 794⟩
    ⟨IrradProcess Data 793⟩
};
```

IrradianceCacheIntegrator 786
IrradianceCacheIntegrator::
  octree 788
IrradProcess 792
IrradProcess::
  GetAverageDirection() 794
IrradProcess::
  GetIrradiance() 794
IrradProcess::
  Successful() 794
Normal 65
Octree::Lookup() 1027
Point 63
RWMutexLock 1039
Scene 22
Spectrum 263
Vector 57

⟨*IrradProcess Data*⟩ ≡ 792
```
Point p;
Normal n;
float minWeight, cosMaxSampleAngleDifference, sumWt;
int nFound;
Spectrum E;
Vector wAvg;
```

The `IrradianceSample` class stores all of the relevant information to represent an irradiance sample—the irradiance value, the position and normal where it was computed, and the average direction of incident illumination. Finally, its `maxDist` member variable stores the distance beyond which the sample absolutely should not be used for shading; its value is set based on information collected when the irradiance sample is computed below.

⟨*IrradianceCacheIntegrator Local Declarations*⟩ +≡
```
struct IrradianceSample {
    ⟨IrradianceSample Constructor⟩
    Spectrum E;
    Normal n;
    Point p;
    Vector wAvg;
    float maxDist;
};
```

Each time the `IrradProcess` callback method is called by `Octree::Lookup()`, it is passed a pointer to an irradiance sample from the octree. The method computes an error term that tries to estimate the error from using the sample at the shading point, which is compared to user-supplied error limits; if the sample passes the test, it's added to a running weighted sum of samples that contribute to the point being shaded.

⟨*IrradianceCacheIntegrator Method Definitions*⟩ +≡
```
bool IrradProcess::operator()(const IrradianceSample *sample) {
    ⟨Compute estimate error term and possibly use sample 794⟩
    return true;
}
```

Two error terms are computed to determine whether a particular candidate irradiance sample can reasonably be used at the lookup point. The first, `perr`, is based on the ratio of the distance from the lookup point to the point where the sample was computed to the sample's maximum allowed distance value. The maximum distance is set using a few criteria; see the fragment ⟨*Compute irradiance sample's contribution extent and bounding box*⟩ in a few pages for details.

The second error term, `nerr`, is based on the angle between the surface normal of the lookup point and the normal direction used for computing the irradiance sample. The estimated error increases as the angle approaches the maximum allowed angle. For efficiency, an error term based on the cosine of the angle is used, so that only a dot product is needed to compare their orientations.

IrradianceSample 793
IrradProcess 792
Normal 65
Point 63
Spectrum 263
Vector 57

Samples with acceptable error are then added to the weighted averages. Each sample is weighted by one minus its error, thus ensuring that low-error samples have higher contribution to the final interpolated result.

⟨*Compute estimate error term and possibly use sample*⟩ ≡ 793
```
float perr = Distance(p, sample->p) / sample->maxDist;
float nerr = sqrtf((1.f - Dot(n, sample->n)) /
                   (1.f - cosMaxSampleAngleDifference));
float err = max(perr, nerr);
if (err < 1.) {
    float wt = 1.f - err;
    E += wt * sample->E;
    wAvg += wt * sample->wAvg;
    sumWt += wt;
}
```

When octree traversal and candidate sample processing is finished, it is necessary to decide if an acceptable interpolated irradiance value has been computed from the irradiance samples. If the sum of weights isn't greater than a user-supplied minimum weight, then we won't use the interpolated value; doing so improves the overall results by preventing the use of a small number of samples with high error.

⟨*IrradProcess Public Methods*⟩ ≡ 792
```
bool Successful() {
    return sumWt >= minWeight;
}
```

The final interpolated irradiance value is a weighted sum of the irradiance values of the acceptable estimates,

$$E = \frac{\sum_i w_i E_i}{\sum_i w_i}.$$

The average direction is just returned directly; the calling code will normalize it as needed.

⟨*IrradProcess Public Methods*⟩ +≡ 792
```
Spectrum GetIrradiance() const { return E / sumWt; }
Vector GetAverageDirection() const { return wAvg; }
```

IrradianceSample::E 793
IrradianceSample::maxDist 793
IrradianceSample::n 793
IrradianceSample::p 793
IrradianceSample::wAvg 793
IrradProcess::E 793
IrradProcess::minWeight 793
IrradProcess::n 793
IrradProcess::sumWt 793
IrradProcess::wAvg 793
Spectrum 263
Vector 57

15.5.3 ADDING NEW VALUES

If the `interpolateE()` method isn't able to find enough nearby irradiance samples of good enough quality, it computes a new irradiance value and adds it to the cache using the ⟨*Compute irradiance at current point*⟩ and ⟨*Add computed irradiance value to cache*⟩ fragments.

To compute a new irradiance estimate, we need to estimate the value of the integral

$$E(\mathrm{p}, \mathbf{n}) = \int_{\mathcal{H}^2(\mathbf{n})} L_{\mathrm{i}}(\mathrm{p}, \omega_{\mathrm{i}}) \, |\cos \theta_{\mathrm{i}}| \, \mathrm{d}\omega_{\mathrm{i}}. \quad [15.14]$$

Because there is no easy available way here to importance sample based on the distribution of incident radiance, the implementation uses a cosine-weighted distribution of directions. It is then faced with the problem of computing the amount of incident radiance along each one, $L_i(p, \omega_i)$—precisely the problem that all of the other integrators in this chapter address.

In the implementation here, the `IrradianceCacheIntegrator` uses standard path tracing to compute these values.[6] Before it starts tracing rays to compute the irradiance estimate, the integrator initializes a pair of random values to use to scramble the two dimensions of a low-discrepancy point sequence that it will map to cosine-weighted directions over the hemisphere. (See Section 7.4.3 for an explanation of how to randomly scramble point sequences so that a different set of sample values is used each time while still preserving the good distribution properties of the point set.) This method also tracks the minimum distance to the first intersection for all of the rays traced; this value is used in determining how widely the sample can be reused without excessive error.

To compute the irradiance estimate from the radiance values along the sample rays, the implementation uses the standard Monte Carlo estimator:

$$E(p, \mathbf{n}) = \frac{1}{N} \sum_j \frac{L_i(p, \omega_j)|\cos\theta_j|}{p(\omega_j)}.$$

Because the rays are sampled from a cosine-weighted distribution, $p(\omega) = \cos\theta/\pi$, so

$$\frac{1}{N} \sum_j \frac{L_i(p, \omega_j)|\cos\theta_j|}{|\cos\theta_j|/\pi} = \frac{\pi}{N} \sum_j L_i(p, \omega_j).$$

⟨*Compute irradiance at current point*⟩ ≡ 791

```
uint32_t scramble[2] = { rng.RandomUInt(), rng.RandomUInt() };
float minHitDistance = INFINITY;
Vector wAvg(0,0,0);
Spectrum LiSum = 0.f;
for (int i = 0; i < nSamples; ++i) {
    ⟨Sample direction for irradiance estimate ray 796⟩
    ⟨Trace ray to sample radiance for irradiance estimate 796⟩
}
E = (M_PI / float(nSamples)) * LiSum;
```

BSDF::LocalToWorld() 480
CosineSampleHemisphere() 669
INFINITY 1002
IrradianceCacheIntegrator 786
IrradianceCacheIntegrator::nSamples 787
M_PI 1002
RNG::RandomUInt() 1003
Spectrum 263
Vector 57

Choosing the direction for the ray just requires generating the 2D low-discrepancy sample and calling `CosineSampleHemisphere()`. It returns a direction in the canonical reflection coordinate system, with the normal direction mapped to the $+z$ axis. To get a world-space ray direction, the convenient `BSDF::LocalToWorld()` method can be used. This direction may then also need to be flipped so that it lies in the same hemisphere as the normal that was passed in.

6 Using bidirectional path tracing to compute these values (to capture, for example, indirect lighting due to a spotlight shining at a small area on a ceiling) would be a good improvement to the basic algorithm implemented here.

⟨*Sample direction for irradiance estimate ray*⟩ ≡ 795
```
float u[2];
Sample02(i, scramble, u);
Vector w = CosineSampleHemisphere(u[0], u[1]);
RayDifferential r(p, bsdf->LocalToWorld(w), rayEpsilon);
r.d = Faceforward(r.d, ng);
```

Once the ray has been generated, path tracing is performed. The method that implements this computation, `pathL()`, is not included here, since it is essentially a stand-alone implementation of the `PathIntegrator`'s `Li()` method from Section 15.3.4. The returned incident radiance estimate is then added to the radiance sum here. The weighted average direction for incident radiance is computed, scaling the current ray's direction by the luminance of the radiance it carries and computing the sum, and then the distance to the intersection point is used to update the variable that tracks the minimum distance of all of the intersection points for this irradiance estimate.

⟨*Trace ray to sample radiance for irradiance estimate*⟩ ≡ 795
```
Spectrum L = pathL(r, scene, renderer, rng, arena);
LiSum += L;
wAvg += r.d * L.y();
minHitDistance = min(minHitDistance, r.maxt);
```

Now that the irradiance estimate has been computed, we need to compute an upper bound of the region of the scene it may provide values for in order to insert the sample into the octree.

⟨*Add computed irradiance value to cache*⟩ ≡ 791
```
⟨Compute irradiance sample's contribution extent and bounding box 797⟩
⟨Allocate IrradianceSample, get write lock, add to octree 797⟩
wi = wAvg;
```

There are three criteria that determine the region in which an irradiance sample may be valid. The first two criteria are based on pixel spacing constraints. As described above, the pixel spacing criteria ensure that, no matter what, we don't deposit samples at a rate greater than `minSamplePixelSpacing` as measured on the image plane, and that, no matter what, we place a sample at the minimum rate of `maxSamplePixelSpacing`. Those parameters are converted to world-space distances here by multiplying by the provided value of the length of the distance between pixels in world space at the point `pixelSpacing`.

The third criterion is based on the minimum distance to the closest intersection found when computing the irradiance sample. The idea is that if there is a nearby occluding object, then the indirect irradiance is likely to be changing quickly around the sample. For example, near the corner where a wall meets the ceiling, irradiance will in general be changing more rapidly than it will be in the middle of the ceiling. This term captures this effect, reducing the region of applicability for samples that are near occluders.

BSDF::LocalToWorld() 480
CosineSampleHemisphere() 669
Faceforward() 66
PathIntegrator 766
Ray::d 67
Ray::maxt 67
RayDifferential 69
Sample02() 372
Spectrum 263
Spectrum::y() 273
Vector 57

⟨*Compute irradiance sample's contribution extent and bounding box*⟩ ≡ 796
```
float maxDist = maxSamplePixelSpacing * pixelSpacing;
float minDist = minSamplePixelSpacing * pixelSpacing;
float contribExtent = Clamp(minHitDistance / 2.f, minDist, maxDist);
BBox sampleExtent(p);
sampleExtent.Expand(contribExtent);
```

Given the `sampleExtent` bounding box that bounds the sample's valid region, we can allocate an irradiance sample object and add it to the octree. Note that we acquire a write lock to the octree before updating it, thus ensuring that other threads aren't concurrently traversing the tree as it is being modified here.

⟨*Allocate* `IrradianceSample`, *get write lock, add to octree*⟩ ≡ 796
```
IrradianceSample *sample = new IrradianceSample(E, p, ng, wAvg,
                                                contribExtent);
RWMutexLock lock(*mutex, WRITE);
octree->Add(sample, sampleExtent);
```

15.6 PARTICLE TRACING AND PHOTON MAPPING

Photon mapping is another approach for solving the LTE based on a biased algorithm. It handles both glossy and diffuse reflection well; perfectly specular reflection is handled in the usual manner with recursive ray tracing.

Photon mapping is one of a family of *particle-tracing* algorithms, which are based on the idea of constructing paths from the lights where, at each vertex of the path, the amount of incident illumination arriving at the vertex is recorded as an illumination sample.[7] After a certain number of these illumination samples have been computed, a data structure that stores a representation of the distribution of light in the scene is built; at rendering time, this representation is used to compute values of measurements needed to compute the image. Because the particle-tracing step is decoupled from computing the measurements, many measurements may be able to reuse the work done for a single particle path, thus leading to more efficient rendering algorithms.

In this section, we will start by introducing a theory of particle-tracing algorithms and will discuss the conditions that must be fulfilled by a particle-tracing algorithm so that arbitrary measurements can be computed correctly using the particles created by the algorithm. We will then describe an implementation of a photon mapping integrator that uses particles to estimate illumination by interpolating lighting contributions from particles around the point being shaded.

BBox 70
BBox::Expand() 72
Clamp() 1000
IrradianceCacheIntegrator::
maxSamplePixelSpacing 787
IrradianceCacheIntegrator::
minSamplePixelSpacing 787
IrradianceCacheIntegrator::
octree 788
IrradianceSample 793
Octree::Add() 1024
RWMutexLock 1039

7 The particle tracing formalism can also be used to understand the "instant global illumination" approach of Section 15.4. Introducing IGI as a variant of bidirectional path tracing, with light paths reused many times, was an equally accurate and somewhat gentler introduction.

★ 15.6.1 THEORETICAL BASIS FOR PARTICLE TRACING

Particle-tracing algorithms in computer graphics are often explained in terms of packets of energy being shot from the light sources in the scene that deposit energy at surfaces they intersect before scattering in new directions. This is an intuitive way of thinking about particle tracing, but the intuition that it provides doesn't make it easy to answer basic questions about how propagation and scattering affect the particles. For example, does their contribution fall off with squared distance like flux density? Or, which $\cos\theta$ terms, if any, affect particles after they scatter from a surface?

In order to give a solid theoretical basis for particle tracing, we will describe it using a framework introduced by Veach (1997, Appendix 4.A), which instead interprets the stored particle histories as samples from the scene's equilibrium radiance distribution. Under certain conditions on the distribution and weights of the particles, the particles can be used to compute estimates of nearly any measurement based on the light distribution in the scene. In this framework, it is quite easy to answer questions about the details of particle propagation like the ones earlier. After developing this theory here, the remainder of this section will demonstrate its application to photon mapping.

A particle-tracing algorithm generates a set of N samples of illumination at points p_j, on surfaces in the scene

$$(\mathrm{p}_j, \omega_j, \alpha_j),$$

where each sample records incident illumination from direction ω_j and has some weight α_j associated with it (Figure 15.18). We would like to determine the conditions on the weights and distribution of particle positions so that we can use them to correctly compute estimates of arbitrary measurements.

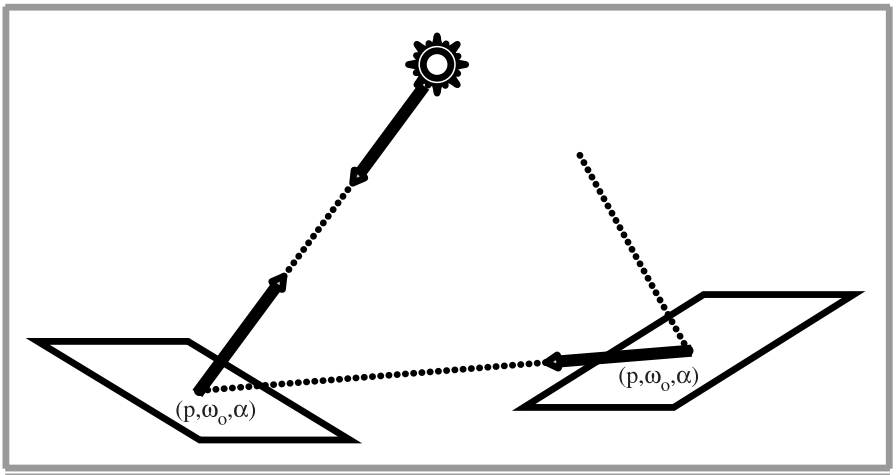


Figure 15.18: When a particle is traced following a path from a light source, an entry in its particle history is recorded at each surface it intersects. Each entry in the history is represented by position p, the direction of the ray it arrived along ω_o, and particle weight α.

Given an importance function $W_e(p, \omega)$ that describes the measurement to be taken, the natural condition we would like to be fulfilled is that the particles should be distributed and weighted such that using them to compute an estimate has the same expected value as the measurement equation for the same importance function:

$$E\left[\frac{1}{N}\sum_{j=1}^{N}\alpha_j W_e(p_j, \omega_j)\right] = \int_A \int_{S^2} W_e(p, \omega) L_i(p, \omega)\, dA\, d\omega. \quad [15.15]$$

For example, we might want to use the particles to compute the total flux on a wall. Using the definition of flux,

$$\Phi = \int_{A_{\text{wall}}} \int_{\mathcal{H}^2(\mathbf{n})} L_i(p, \omega)\, |\cos\theta|\, dA\, d\omega,$$

the following importance function selects the particles that lie on the wall and arrived from the hemisphere around the normal:

$$W_e(p, \omega) = \max((\omega \cdot \mathbf{n}), 0) \times \begin{cases} 1 & p \text{ is on wall surface} \\ 0 & \text{otherwise.} \end{cases}$$

If the conditions on the distribution of particle weights and positions are true for arbitrary importance functions such that Equation (15.15) holds, then the flux estimate can be computed directly as just a sum of the particle weights for the particles on the wall multiplied by the $(\omega \cdot \mathbf{n})$ term. If we want to estimate flux over a different wall, a subset of the original wall, and so on, we only need to recompute the weighted sum with an updated importance function. The particles and weights can be reused, and we have an unbiased estimate for all of these measurements. (The estimates will be correlated, however, which is potentially a source of artifacts.)

To see how to generate and weight particles that fulfill these conditions, consider the task of evaluating the measurement equation integral

$$\begin{aligned}&\int_A \int_{S^2} W_e(p_0, \omega)\, L(p_0, \omega)\, d\omega\, dA(p_0) \\ &\quad = \int_A \int_A W_e(p_0 \to p_1)\, L(p_1 \to p_0)\, G(p_0 \leftrightarrow p_1)\, dA(p_0)\, dA(p_1),\end{aligned}$$

where the importance function W_e that describes the measurement is a black box and thus cannot be used to drive the sampling of the integral at all. We can still compute an estimate of the integral with Monte Carlo integration, but must sample a set of points p_0 and p_1 from all of the surfaces in the scene, using some sampling distribution that doesn't depend on W_e (e.g., by uniformly sampling points by surface area).

By expanding the LTE in the integrand and applying the standard Monte Carlo estimator for N samples, we can find the estimator for this measurement,

$$E\left[\frac{1}{N}\sum_{i=1}^{N} W_e(p_{i,0} \to p_{i,1}) \left\{\frac{L(p_{i,1} \to p_{i,0}) G(p_{i,0} \leftrightarrow p_{i,1})}{p(p_{i,0}) p(p_{i,1})}\right\}\right].$$

We can further expand out the L term into the sum over paths and use the fact that $E[ab] = E[aE[b]]$ and the fact that for a particular sample, the expected value

$$E\left[\frac{L(\mathrm{p}_{i,1} \to \mathrm{p}_{i,0})}{p(\mathrm{p}_{i,0})}\right]$$

can be written as a finite sum of n_i terms in just the same way that we generated a finite set of weighted path vertices for path tracing. If the sum is truncated with Russian roulette such that the probability of continuing the sum after j terms is $q_{i,j}$, then the jth term of the ith sample has contribution

$$\frac{L_\mathrm{e}(\mathrm{p}_{n_i} \to \mathrm{p}_{n_i-1})}{p(\mathrm{p}_{n_i})} \prod_{j=1}^{n_i-1} \frac{1}{q_{i,j}} \frac{f(\mathrm{p}_{i,j+1} \to \mathrm{p}_{i,j} \to \mathrm{p}_{i,j-1}) G(\mathrm{p}_{i,j+1} \leftrightarrow \mathrm{p}_{i,j})}{p(\mathrm{p}_{i,j})}.$$

Looking back at Equation (15.15), we can see that this quantity gives the appropriate value of the particle weights if the particle-generating paths are sampled from a distribution over area:

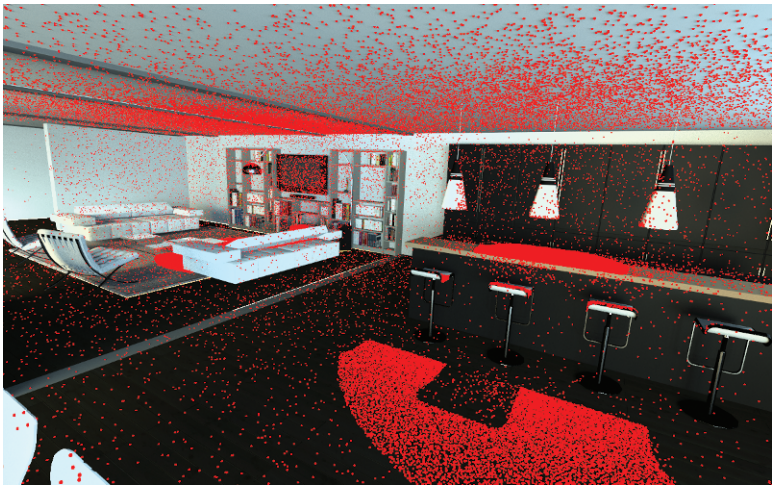
$$\alpha_{i,j} = \frac{L_\mathrm{e}(\mathrm{p}_{n_i} \to \mathrm{p}_{n_i-1})}{p(\mathrm{p}_{n_i})} \prod_{j=1}^{n_i-1} \frac{1}{q_{i,j}} \frac{f(\mathrm{p}_{i,j+1} \to \mathrm{p}_{i,j} \to \mathrm{p}_{i,j-1}) G(\mathrm{p}_{i,j+1} \leftrightarrow \mathrm{p}_{i,j})}{p(\mathrm{p}_{i,j})}. \quad \text{(15.16)}$$

Note that we have the freedom to generate a set of particles with these weights in all sorts of different ways. Although the natural approach is to start from points on lights and incrementally sample paths using the BSDFs at the path vertices, similar to how the path-tracing integrator generates paths (starting here from the light, rather than from the camera), we could generate them with any number of different sampling strategies, so long as there is nonzero probability of generating a particle at any point where W_e is nonzero and the particle weights are computed appropriately for the sampling distribution used.

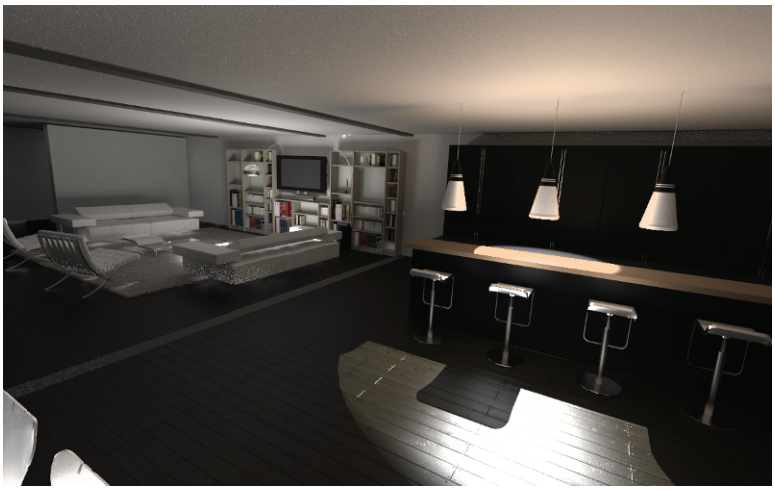
If we only had a single measurement to make, it would be better if we used information about W_e and could compute the estimate more intelligently, since the general particle-tracing approach described here may generate many useless samples if W_e only covers a small subset of the points on scene objects. If we will be computing many measurements, however, the key advantage that particle tracing brings is that we can generate the samples and weights once and can then reuse them over a large number of measurements, potentially computing results much more efficiently than if the measurements were all computed from scratch.

15.6.2 PHOTON INTEGRATOR

The photon mapping integrator traces particles into the scene and interpolates among particles to approximate the incident illumination at shading points. For consistency with other descriptions of the algorithm, we will refer to particles generated for photon mapping as photons. The integrator uses a kd-tree data structure to store the photons; this allows it to quickly find the photons around the point being shaded. The kd-tree is referred to as the *photon map*. Because the kd-tree is decoupled from the scene geometry, this algorithm isn't limited to a particular type of geometric representation (in contrast to using a texture map defined over shapes' (u, v) parameterizations to store illumina-



(a)



(b)

Figure 15.19: Rendering the Contemporary House Scene with Photon Mapping. (a) Locations of photons stored in the photon map. More photons are present in areas with brighter illumination. (b) Scene rendered using photon mapping. *(Model courtesy of Florent Boyer.)*

tion, for instance). Figure 15.19(a) shows the distribution of photons for a view of the contemporary house scene, and Figure 15.19(b) shows the scene rendered with photon mapping. Note that there is a greater density of photons in areas with more illumination.

Photon mapping partitions the LTE in a few ways that make it easier to adjust the quality of the results computed. For example, particles from the lights are characterized as being one of three types: direct illumination (light that has arrived directly at a surface, without any scattering), caustic illumination (light that has arrived at a nonspecular surface after

interacting with one or more specular surfaces), and indirect illumination (all other types of illumination). Thanks to this partitioning, the integrator can apply specific strategies to each type of illumination. In the implementation here, direct illumination is handled entirely with conventional direct lighting algorithms and caustics are always rendered directly with the photons in the photon map.

General indirect illumination is usually rendered using *final gathering*, where one bounce of path tracing is performed and the photon map is then used to compute reflection at the intersection points. (The photon map can also be used directly to approximate indirect illumination at shaded points without final gathering, though the visual results generally have artifacts.)

Quite a few different parameters control the operation of the integrator. The user must specify a desired number of photons of each type—caustic and indirect—to store in each of the two respective photon maps. Using more photons increases the quality of results but takes more time and memory. Because this integrator interpolates nearby photons to estimate illumination at the shading point, the user can also set how many photons are used for the interpolation. The more photons that are used, the smoother the illumination estimate will be, since a larger number of photons will be used to reconstruct it. If too many are used, the result will tend to be too blurry, while too few gives a splotchy appearance. Interpolating 50 to 100 photons is often a good choice. See the "Further Reading" section at the end of this chapter for references to research on more accurate methods of filtering photon contributions for reconstructing illumination.

We won't include the implementation of the `PhotonIntegrator` constructor here, since it just initializes its member variables from the parameters passed to it. Its member variables are the following:

- `nCausticPhotonsWanted` and `nIndirectPhotonsWanted` give the total number of photons to try to store for each category of illumination stored in the photon map. During preprocessing before rendering, ray paths will be followed until either the requested number of photons have been deposited or the system detects that it's not successfully depositing photons—for example, no caustic light-carrying paths will be found in a scene with no specular surfaces.
- `nLookup` gives the total number of photons to try to use for the interpolation step.
- `maxDistSquared` gives the maximum allowed squared distance from the point where exitant radiance is being computed to a photon that can be used for the interpolation there. If its value is too large, the integrator will waste time searching to find photons in dark regions; if it's too small, it may not be able to find `nLookup` nearby photons, leading to an overly splotchy result.
- `maxSpecularDepth` tracks the maximum path depth of specular reflection, similar to the Whitted integrator, and `maxPhotonDepth` can similarly limit the length of paths traced during the photon map construction step.
- `finalGather` controls if final gathering is used for indirect lighting rather than using the indirect photon map directly; if final gathering is enabled, `gatherSamples` controls the number of samples taken—64 to 128 samples generally work well if a single image sample per pixel is taken; as more pixel samples are taken, the number of gather samples for each one can be reduced correspondingly. If final gathering is enabled, then `cosGatherAngle` controls how the photons are used for importance

sampling directions for the final gather rays; they represent information about the directional distribution of incident illumination that can be used to improve the distribution of these rays.

⟨*PhotonIntegrator Private Data*⟩ ≡
```
uint32_t nCausticPhotonsWanted, nIndirectPhotonsWanted, nLookup;
float maxDistSquared;
int maxSpecularDepth, maxPhotonDepth;
bool finalGather;
int gatherSamples;
float cosGatherAngle;
```

The `PhotonIntegrator::RequestSamples()` method is similar to implementations in other integrators. It requests samples for multiple importance sampling for direct lighting and, if final gathering has been enabled, a well-distributed set of samples for sampling the BSDF for the final gather step.

⟨*PhotonIntegrator Method Definitions*⟩ ≡
```
void PhotonIntegrator::RequestSamples(Sampler *sampler, Sample *sample,
        const Scene *scene) {
    ⟨Allocate and request samples for sampling all lights 743⟩
    ⟨Request samples for final gathering 803⟩
}
```

⟨*PhotonIntegrator Private Data*⟩ +≡
```
⟨Declare sample parameters for light source sampling 743⟩
```

In the implementation to follow, final gathering is done by sampling the BSDF to find directions for half of the final gather rays and sampling the indirect illumination distribution, as represented by the nearby photons' directions, for the other half. The `PhotonIntegrator::RequestSamples()` method therefore needs to request two sets of samples for final gathering; one will be used for the BSDF samples and the other for the samples taken using the nearby photons to define a sampling distribution.

⟨*PhotonIntegrator Private Data*⟩ +≡
```
BSDFSampleOffsets bsdfGatherSampleOffsets, indirGatherSampleOffsets;
```

⟨*Request samples for final gathering*⟩ ≡ 803
```
if (finalGather) {
    gatherSamples = max(1, gatherSamples/2);
    if (sampler) gatherSamples = sampler->RoundSize(gatherSamples);
    bsdfGatherSampleOffsets = BSDFSampleOffsets(gatherSamples, sample);
    indirGatherSampleOffsets = BSDFSampleOffsets(gatherSamples, sample);
}
```

BSDFSampleOffsets 706
PhotonIntegrator::finalGather 803
PhotonIntegrator::gatherSamples 803
Sample 343
Sampler 340
Sampler::RoundSize() 344
Scene 22

15.6.3 BUILDING THE PHOTON MAPS

When the `Preprocess()` method is called, particle paths are followed through the scene until the integrator has accumulated the desired number of particle histories to build the photon maps. Three separate photon maps are created: one to store photons that represent the direct illumination, one for indirect lighting, and one for caustics. (The

direct illumination photons are used only during this preprocessing step and are freed at the end of this method.) At each intersection of the path with an object, a weighted particle contribution is stored if the map for the corresponding type of illumination is not yet full. This phase of computation is executed in parallel; the concurrently running tasks must therefore carefully coordinate their updates of the shared data structures.

```
⟨PhotonIntegrator Method Definitions⟩ +≡
  void PhotonIntegrator::Preprocess(const Scene *scene,
          const Camera *camera, const Renderer *renderer) {
      if (scene->lights.size() == 0) return;
      ⟨Declare shared variables for photon shooting 804⟩
      ⟨Compute light power CDF for photon shooting 805⟩
      ⟨Run parallel tasks for photon shooting⟩
      ⟨Build kd-trees for indirect and caustic photons 813⟩
      ⟨Precompute radiance at a subset of the photons 815⟩
  }
```

A number of shared variables will be updated by the photon shooting tasks as they execute; references to them are passed into the tasks. Updates to these variables by the tasks must be protected with the mutex declared here. There are three arrays of `Photon` objects, which represent illumination samples (the contents of the `Photon` structure will be defined shortly). There is also an array of `RadiancePhotons`; they represent outgoing radiance at points in the scene, in contrast to the regular photons, which effectively represent incident illumination. The `RadiancePhotons` are used to accelerate final gathering computations; details of their use will be explained later in this section.

`abortTasks` is only set to `true` if one of the tasks determines that not enough photons are being stored with respect to the number of paths being traced; in this case, when the other tasks notice that it has been set, they exit.

Finally, the total number of paths started from the lights by all of the tasks is maintained in the `nshot` variable. In the end, this value may be larger or smaller than the sizes of the respective arrays of photons. On the one hand, it may be necessary to start more paths than photons are stored—for example, photons may leave the scene without intersecting any objects. On the other hand, each path may contribute multiple entries to the particle history as it bounces around the scene.

```
⟨Declare shared variables for photon shooting⟩ ≡ 804
  Mutex *mutex = Mutex::Create();
  int nDirectPaths = 0;
  vector<Photon> causticPhotons, directPhotons, indirectPhotons;
  vector<RadiancePhoton> radiancePhotons;
  bool abortTasks = false;
  uint32_t nshot = 0;
```

The `Photon` structure stores just enough information to record a photon's contribution—the position where it hit the surface, its weight, and the direction it arrived from. Because hundreds of thousands or millions of photons may be stored, using a more compact

Camera 302
Mutex 1038
Photon 805
PhotonIntegrator 802
RadiancePhoton 815
Renderer 24
Scene 22
Scene::lights 23

photon representation than the one used here can be worthwhile. To keep the implementation here simple, however, we will just use the straightforward representation below.

⟨*PhotonIntegrator Local Declarations*⟩ ≡
```
struct Photon {
    Photon(const Point &pp, const Spectrum &wt, const Vector &w)
        : p(pp), alpha(wt), wi(w) { }
    Point p;
    Spectrum alpha;
    Vector wi;
};
```

Recall that it is both the distribution and the weights of the collection of particles in a scene that together represent the equilibrium radiance distribution. For example, the light on a surface with a relatively large amount of illumination arriving at it could be represented by a small number of particles each with large weights, a large number of particles with smaller weights, or something between these two extremes. As far as the theory underlying the method, any of these representations is an equivalently reasonable representation of the scene's radiance distribution.

In practice, the photon mapping algorithm works best if all of the photons have the same (or similar) weights, and if it is only their varying density throughout the scene that represents the variation in illumination. If the photon weights have substantial variation, there can be unpleasant image artifacts. If one photon takes on a much larger weight than the others, it is easy to see the region of the scene where that photon contributes to the interpolated radiance: there is a bright circular artifact in that area.

Therefore, we'd like to shoot more photons from the brighter lights so that the initial weights of photons leaving all lights will be of similar magnitudes, and thus, the light to start each path from is chosen according to a PDF defined by the lights' respective power. Thus, it is a greater number of photons from the brighter lights that accounts for their greater contributions to illumination in the scene rather than a lower number of photons but with larger weights than the other lights. To implement this approach, a discrete PDF is computed to choose lights to shoot particles from based on each light's power.

⟨*Compute light power CDF for photon shooting*⟩ ≡ 804
```
Distribution1D *lightDistribution = ComputeLightSamplingCDF(scene);
```

The `PhotonShootingTask` handles the parallel creation of the photon maps. We won't include the code that launches the tasks, as it's essentially the same as other task launch code, and we've elided the declaration of the `PhotonShootingTask` class, which mostly just stores references to variables declared here and the parameters to the `Preprocess()` method so that they are available in the task's `Run()` method.

The `Run()` method stores photons in local arrays as it traces paths through the scene. After a few thousand paths have been followed, it acquires the mutex and updates the shared photon data structures. Compared to acquiring the mutex and updating the data structures each time a task deposits a photon, this approach is more efficient by amortizing the overhead of acquiring the mutex over more work.

ComputeLightSamplingCDF() 709
Distribution1D 648
Photon 805
Point 63
Spectrum 263
Vector 57

⟨*PhotonIntegrator Method Definitions*⟩ +≡
```
void PhotonShootingTask::Run() {
    ⟨Declare local variables for PhotonShootingTask 806⟩
    while (true) {
        ⟨Follow photon paths for a block of samples 807⟩
        ⟨Merge local photon data with data in PhotonIntegrator 812⟩
        ⟨Exit task if enough photons have been found 813⟩
    }
}
```

In addition to the local arrays for storing photons before they're merged to the shared arrays declared in `PhotonIntegrator::Preprocess()`, a `MemoryArena` and a `RNG` are needed. Note that the random number generator is seeded with a different value for each task so that each task follows different photon paths through the scene. `totalPaths` tracks the total number of photon paths that have been followed by this task, and `causticDone` and `indirectDone` are `false` until the requested number of photons of their respective types have been stored; they start out as `true` if zero of the corresponding type is needed.

⟨*Declare local variables for* `PhotonShootingTask`⟩ ≡ 806
```
MemoryArena arena;
RNG rng(31 * taskNum);
vector<Photon> localDirectPhotons, localIndirectPhotons, localCausticPhotons;
vector<RadiancePhoton> localRadiancePhotons;
uint32_t totalPaths = 0;
bool causticDone = (integrator->nCausticPhotonsWanted == 0);
bool indirectDone = (integrator->nIndirectPhotonsWanted == 0);
```

Six sample values are needed to start each path: one to select which light source to start the path from and five more for the `Light::Sample_L()` method that generates rays leaving the light. The fact that tasks are simultaneously generating samples with multiple threads of execution makes the sample generation problem more challenging than it would be otherwise. With a single-threaded implementation, we might use a Halton sequence for generating these samples, taking advantage of the fact that no matter when we stop generating more samples, in the aggregate, the samples would be well distributed. In a parallel implementation, we can't directly use the Halton sequence in each thread, since then all threads would generate the same set of photon paths! Therefore, the implementation here uses the randomized variant of the Halton sequence that was introduced in Section 7.4.2, using the `PermutedHalton` utility class, which is given the number of dimensions for which samples are needed as well as the uniquely seeded `RNG` for this task.

⟨*Declare local variables for* `PhotonShootingTask`⟩ +≡ 806
```
PermutedHalton halton(6, rng);
```

The task now traces `blockSize` paths from the lights, storing the resulting photons in the local arrays. Only after this phase has finished will the task acquire the mutex and merge its photons with the global data structures. To create a new path, the integrator samples a ray from one of the lights in the scene and then follows the path, recording

Light::Sample_L() 608
MemoryArena 1015
PermutedHalton 367
Photon 805
PhotonIntegrator::
nCausticPhotonsWanted 803
PhotonIntegrator::
nIndirectPhotonsWanted 803
RadiancePhoton 815
RNG 1003

particle intersections as it bounces around the scene. The `alpha` variable represents the particle weight from Equation (15.16) and is incrementally updated to store the path contribution at each vertex. After path termination, the BSDF memory allocated for the path and maintained by the `MemoryArena` is freed before going on to start the next one.

⟨*Follow photon paths for a block of samples*⟩ ≡ 806
```
const uint32_t blockSize = 4096;
for (uint32_t i = 0; i < blockSize; ++i) {
    float u[6];
    halton.Sample(++totalPaths, u);
    ⟨Choose light to shoot photon from 807⟩
    ⟨Generate photonRay from light source and initialize alpha 808⟩
    if (!alpha.IsBlack()) {
        ⟨Follow photon path through scene and record intersections 808⟩
    }
    arena.FreeAll();
}
```

Which light to start the path from is determined using the CDF based on light power computed previously. Here, the first component of the 6D sample from the randomly permuted Halton sequence, `u[0]`, is used.

⟨*Choose light to shoot photon from*⟩ ≡ 807
```
float lightPdf;
int lightNum = lightDistribution->SampleDiscrete(u[0], &lightPdf);
const Light *light = scene->lights[lightNum];
```

After the light has been chosen, its `Sample_L()` method is used to sample an outgoing ray. Given a light, a ray from the light source is sampled and its α value is initialized with

$$\alpha = \frac{|\cos \omega_0| \, L_e(\mathrm{p}_0, \omega_0)}{p(\mathrm{p}_0, \omega_0)},$$

where $p(\mathrm{p}_0, \omega_0)$ is the product of the probability for sampling this particular light and the product of the area and directional densities for sampling this particular ray leaving the light. The remaining five dimensions of the sample from the permuted Halton sequence are used for the sample values.

Note that it isn't possible to guarantee that all of the rays sampled by a light will have the same weights—some variation in photon weights may remain in spite of the efforts to choose lights according to their total power. For example, an area light with a texture map defining an illumination distribution might choose points uniformly over the light's surface rather than according to a PDF that matched the texture map's variation. In such a case, the weights for the rays from the lights would still be varying. Thus, though the implementation here does its best to create photons with equal weights, it's not possible to guarantee this property in all cases.

Distribution1D::SampleDiscrete() 650
Light 606
MemoryArena 1015
MemoryArena::FreeAll() 1017
PermutedHalton::Sample() 368
Spectrum::IsBlack() 265

⟨*Generate* photonRay *from light source and initialize* alpha⟩ ≡ 807
```
RayDifferential photonRay;
float pdf;
LightSample ls(u[1], u[2], u[3]);
Normal Nl;
Spectrum Le = light->Sample_L(scene, ls, u[4], u[5],
                              time, &photonRay, &Nl, &pdf);
if (pdf == 0.f || Le.IsBlack()) continue;
Spectrum alpha = (AbsDot(Nl, photonRay.d) * Le) / (pdf * lightPdf);
```

Now the integrator can start following the path through the scene, updating α after each scattering event and recording photons at the path vertices. The specularPath variable records whether the current path has only intersected perfectly specular surfaces after leaving the light source, which indicates that the path is a caustic path. Its value isn't used at the first intersection (direct lighting), so we can safely initialize its value to true here knowing that it will be set to false after any nonspecular bounce.

⟨*Follow photon path through scene and record intersections*⟩ ≡ 807
```
bool specularPath = true;
Intersection photonIsect;
int nIntersections = 0;
while (scene->Intersect(photonRay, &photonIsect)) {
    ++nIntersections;
    ⟨Handle photon/surface intersection 808⟩
    ⟨Sample new photon ray direction 810⟩
}
```

Given a photon–surface intersection, the particle weight must be updated to account for attenuation due to participating media from the last path vertex to this one. The photon may be stored in the appropriate array of photons, depending on whether more photons of the corresponding type are still required. However, if the current hit is a completely specular surface, there's no need to record it in any photon map, since we do not use photons for rendering reflections from specular surfaces.

⟨*Handle photon/surface intersection*⟩ ≡ 808
```
alpha *= renderer->Transmittance(scene, photonRay, NULL, rng, arena);
BSDF *photonBSDF = photonIsect.GetBSDF(photonRay, arena);
BxDFType specularType = BxDFType(BSDF_REFLECTION |
                        BSDF_TRANSMISSION | BSDF_SPECULAR);
bool hasNonSpecular = (photonBSDF->NumComponents() >
                       photonBSDF->NumComponents(specularType));
Vector wo = -photonRay.d;
if (hasNonSpecular) {
    ⟨Deposit photon at surface 810⟩
}
if (nIntersections >= integrator->maxPhotonDepth) break;
```

AbsDot() 61
BSDF 478
BSDF::NumComponents() 479
BSDF_REFLECTION 428
BSDF_SPECULAR 428
BSDF_TRANSMISSION 428
BxDFType 428
Intersection 186
Intersection::GetBSDF() 484
Light::Sample_L() 608
LightSample 710
Normal 65
PhotonIntegrator::
maxPhotonDepth 803
Ray::d 67
RayDifferential 69
Renderer::Transmittance() 25
Scene::Intersect() 23
Spectrum 263
Spectrum::IsBlack() 265
Vector 57

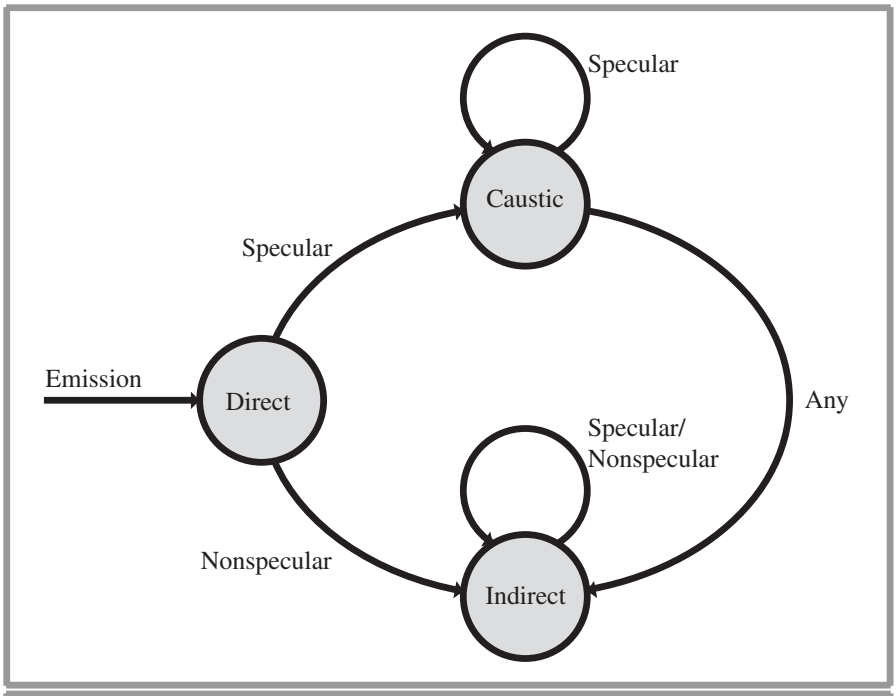


Figure 15.20: Finite-State Machine for Determining Which Photon Map to Update. An illustration of the transitions photons make from one state to another depending on the type of surface scattering they undergo at path vertices.

If this is the first intersection found after the particle has left the light source, then the photon represents direct illumination. Otherwise, if it has only reflected from specular surfaces before arriving at the current intersection point, it must be a caustic photon. Any other case—a path that only hit nonspecular surfaces or a path that hit a series of specular surfaces before scattering from nonspecular surfaces—represents indirect illumination. The finite-state machine diagram in Figure 15.20 illustrates the possibilities. The nodes of the graph show which of the photon maps should be updated for a photon in that state, and the edges describe whether the photon has scattered from a specular or a nonspecular BSDF component at its previous intersection. The code fragment here only includes the code for updating the caustic photon map; the code for direct and indirect photons is analogous and not included here.

As the photon maps are being populated, a random subset of photons is selected to also be `RadiancePhotons`, which store a precomputed outgoing radiance value. The fragment that creates them, ⟨*Possibly create radiance photon at photon intersection point*⟩, will be defined in a few pages, in the section that discusses radiance photons.

RadiancePhoton 815

⟨*Deposit photon at surface*⟩ ≡ 808
```
Photon photon(photonIsect.dg.p, alpha, wo);
bool depositedPhoton = false;
if (specularPath && nIntersections > 1) {
    if (!causticDone) {
        depositedPhoton = true;
        localCausticPhotons.push_back(photon);
    }
}
else {
    ⟨Deposit either direct or indirect photon⟩
}
⟨Possibly create radiance photon at photon intersection point 814⟩
```

Having recorded the particle's contribution in one of the photon maps (or ignoring it if that map type was full), the integrator needs to choose a new outgoing direction from the intersection point and update the α value to account for the effect of scattering of the incident illumination. Equation (15.15) shows how to incrementally update the particle weight after a scattering event: Given some weight $\alpha_{i,j}$ that represents the weight for the jth intersection of the ith particle history, after a scattering event where a new vertex $\mathrm{p}_{i,j+1}$ has been sampled, the weight should be set to be

$$\alpha_{i,j+1} = \alpha_{i,j} \frac{1}{q_{i,j+1}} \frac{f(\mathrm{p}_{i,j+1} \rightarrow \mathrm{p}_{i,j} \rightarrow \mathrm{p}_{i,j-1}) G(\mathrm{p}_{i,j+1} \leftrightarrow \mathrm{p}_{i,j})}{p(\mathrm{p}_{i,j+1})}.$$

As with the path-tracing integrator, there are a number of reasons to choose the next vertex in the path by sampling the BSDF's distribution at the intersection point to get a direction ω' and tracing a ray in that direction, rather than directly sampling by area on the scene surfaces. Therefore, we again apply the Jacobian to account for this change in measure, all of the terms in G except for a single $|\cos\theta|$ cancel out, and the expression is

$$\alpha_{i,j+1} = \alpha_{i,j} \frac{1}{q_{i,j+1}} \frac{f(\mathrm{p}, \omega, \omega')|\cos\theta'|}{p(\omega')}. \quad [15.17]$$

The code here uses uniform random numbers for these samples, since the advantages of further low-discrepancy points are mostly lost with increasing numbers of bounces: the variation in paths introduced by complex scene geometry overwhelms any well-distributed properties of the sample values.

⟨*Sample new photon ray direction*⟩ ≡ 808
```
Vector wi;
float pdf;
BxDFType flags;
Spectrum fr = photonBSDF->Sample_f(wo, &wi, BSDFSample(rng),
                                   &pdf, BSDF_ALL, &flags);
if (fr.IsBlack() || pdf == 0.f) break;
Spectrum anew = alpha * fr *
    AbsDot(wi, photonBSDF->dgShading.nn) / pdf;
⟨Possibly terminate photon path with Russian roulette 811⟩
specularPath &= ((flags & BSDF_SPECULAR) != 0);
```

AbsDot() 61
BSDF::dgShading 479
BSDF::Sample_f() 706
BSDFSample 705
BSDF_ALL 428
BSDF_SPECULAR 428
BxDFType 428
DifferentialGeometry::nn 102
DifferentialGeometry::p 102
Intersection::dg 186
Intersection::rayEpsilon 186
Photon 805
RayDifferential 69
Spectrum 263
Spectrum::IsBlack() 265
Vector 57

```
if (indirectDone && !specularPath) break;
photonRay = RayDifferential(photonIsect.dg.p, wi, photonRay,
                            photonIsect.rayEpsilon);
```

The photon scattering step needs to be implemented carefully in order to keep the photon weights as similar to each other as possible. A method that gives distribution of photons where all have exactly equal weights was suggested by Jensen (2001, Section 5.2). First, the reflectance is computed at the intersection point. A random decision is then made whether or not to continue the photon's path with probability proportional to this reflectance. If the photon continues, its scattered direction is found by sampling from the BSDF's distribution, but it continues with its weight unchanged except for adjusting the spectral distribution based on the surface's color. Thus, a surface that reflects very little light will reflect few of the photons that reach it, but those that are scattered will continue on with unchanged contributions and so forth.

This particular approach isn't possible in `pbrt` due to a subtle implementation detail (a similar issue held for light source sampling previously as well): in `pbrt`, the `BxDF` interfaces are written so that the distribution used for importance sampling BSDFs doesn't necessarily have to perfectly match the actual distribution of the function being sampled. It is all the better if it does, but for many complex BSDFs exactly sampling from its distribution is difficult or impossible.

Therefore, here we will use an approach that generally leads to a similar result but offers more flexibility: at each intersection point, an outgoing direction is sampled with the BSDF's sampling distribution, and the photon's updated weight $\alpha_{i,j+1}$ is computed using Equation (15.17). Then, the ratio of the luminance of $\alpha_{i,j+1}$ to the luminance of the photon's old weight $\alpha_{i,j}$ is used to set the probability of continuing the path after applying Russian roulette.

The probability is thus set so that if the photon's weight is significantly decreased at the scattering point, the termination probability will be high and if the photon's weight is essentially unchanged, the termination probability is low. In particular, the termination probability is chosen in a way such that if the photon continues, after its weight has been adjusted for the possibility of termination, its luminance will be the same as it was before scattering. It is easy to verify this property from the fragment below. (This property actually doesn't hold for the case where $\alpha_{i,j+1} > \alpha_{i,j}$, as can happen when the ratio of the BSDF's value and the PDF is greater than one.)

⟨*Possibly terminate photon path with Russian roulette*⟩ ≡ 810

```
float continueProb = min(1.f, anew.y() / alpha.y());
if (rng.RandomFloat() > continueProb)
    break;
alpha = anew / continueProb;
```

Updating the Shared Data Structures

PhotonIntegrator 802
RNG::RandomFloat() 1003
Spectrum::y() 273

After `blockSize` photon paths have been followed, the task acquires the mutex so that it can safely modify the shared photon map data structures in the `PhotonIntegrator`. The shared `nshot` value is updated to reflect the additional paths followed by this task, and the photons stored in the local arrays are copied into the shared arrays.

⟨*Merge local photon data with data in* PhotonIntegrator⟩ ≡ 806
```
{ MutexLock lock(mutex);
⟨Give up if we're not storing enough photons⟩
nshot += blockSize;
⟨Merge indirect photons into shared array 812⟩
⟨Merge direct, caustic, and radiance photons into shared array⟩
}
```

The task may find that it and the other tasks have generated many paths while storing few to no photons of some types. In this case, it gives up and signals the other tasks to exit by setting `abortTasks` to `true`. (And, if it detects that another task has set this variable, it exits immediately.) This situation may happen, for example, if it was trying to populate a caustic map but there weren't any specular objects in the scene to create caustic paths. This task is handled by the ⟨*Give up if we're not storing enough photons*⟩ fragment, which is straightforward and not included here.

In addition to the photons themselves, the integrator needs to know many paths from the lights were followed to generate the collections of stored photons. These values are necessary for the density estimation algorithm used to interpolate among photons around a point being shaded; they're stored in the `nCausticPaths` and `nIndirectPaths` `PhotonIntegrator` member variables.

⟨*PhotonIntegrator Private Data*⟩ +≡
```
int nCausticPaths, nIndirectPaths;
```

After adding the photons to the shared array, the implementation also checks to see if the additional photons it has added are enough such that the number of requested photons has been found. If so, it updates its `indirectDone` (or, respectively, `causticDone`) variable to indicate that no more photons of that type are needed. There is a subtlety in the implementation here: only the local variable `indirectDone` and not the shared array size is checked before adding the photons to the shared array, so the shared arrays may already have enough photons in them without the additional ones. The rationale is that, since the task here has already gone through the trouble of generating the photons, it might as well store them rather than just throwing them away if enough have already been found. (The fragments that merge the other types of photons are essentially the same and not included here.)

⟨*Merge indirect photons into shared array*⟩ ≡ 812
```
if (!indirectDone) {
    integrator->nIndirectPaths += blockSize;
    for (uint32_t i = 0; i < localIndirectPhotons.size(); ++i)
        indirectPhotons.push_back(localIndirectPhotons[i]);
    localIndirectPhotons.erase(localIndirectPhotons.begin(),
                               localIndirectPhotons.end());
    if (indirectPhotons.size() >= integrator->nIndirectPhotonsWanted)
        indirectDone = true;
}
```

MutexLock 1039
PhotonIntegrator::
nIndirectPaths 812
PhotonIntegrator::
nIndirectPhotonsWanted 803

Once enough of both types of photons have been stored, the task can exit.

⟨*Exit task if enough photons have been found*⟩ ≡ 806
```
if (indirectDone && causticDone)
    break;
```

After all of the photon shooting tasks exit, the `PhotonIntegrator::Preprocess()` method continues execution. It next creates a kd-tree for each of the three photon maps. (The `KdTree` template class used is described in Section A.8 of Appendix A.) Because the direct lighting photons are only used during the preprocessing step and not during scene rendering, their kd-tree is stored in a local variable rather than a `PhotonIntegrator` member variable.

⟨*Build kd-trees for indirect and caustic photons*⟩ ≡ 804
```
KdTree<Photon> *directMap = NULL;
if (directPhotons.size() > 0)
    directMap = new KdTree<Photon>(directPhotons);
if (causticPhotons.size() > 0)
    causticMap = new KdTree<Photon>(causticPhotons);
if (indirectPhotons.size() > 0)
    indirectMap = new KdTree<Photon>(indirectPhotons);
```

⟨*PhotonIntegrator Private Data*⟩ +≡
```
KdTree<Photon> *causticMap;
KdTree<Photon> *indirectMap;
```

Precomputed Radiance Values

If final gathering is being performed, a few tens of final gather rays are traced over the hemisphere at each point passed to `PhotonIntegrator::Li()` in order to sample incident indirect illumination. In order to compute the incident radiance arriving at each final gather ray's origin, it is necessary to compute the outgoing radiance at the intersection point of each of the final gather rays. One option would be to use the `LPhoton()` function (defined later in this section), which searches the kd-tree to find the nearest tens or hundreds of photons around the intersection point and then interpolates among them, though the computational overhead of doing this for each of the final gather rays can be substantial.

We can instead take advantage of the fact that outgoing radiance values computed with the photon map at the final gather ray intersection points don't need to be as accurate as the radiance values computed with the photon map for directly visible points. The `PhotonIntegrator` selects a subset of the initial photons and creates a separate `RadiancePhoton` for each of them. The `RadiancePhoton` stores an exitant radiance value that is computed with the surface's reflectance and the incident radiance distribution at the photon's position. When the intersection point of a final gather ray is found, the *single* nearest `RadiancePhoton` is found and its radiance value is used to approximate the outgoing radiance at the intersection point. This is substantially more efficient than doing a full photon map lookup and photon interpolation.

Figure 15.21 shows an image of a scene rendered using the radiance photons directly, effectively showing the scene as "seen" by the final gathering rays. Note that the illumination appears as a smoothed but generally accurate approximation to the actual illumination.

KdTree 1029
LPhoton() 831
Photon 805
PhotonIntegrator 802
PhotonIntegrator::
causticMap 813
PhotonIntegrator::
indirectMap 813
PhotonIntegrator::Li() 817
RadiancePhoton 815

Figure 15.21: Rendering with Radiance Photons. Scene rendered directly using radiance values from radiance photons to shade visible surfaces. The radiance photons do a reasonable job of capturing an approximation of the outgoing radiance from surfaces in the scene. When used to compute radiance along the final gathering rays, the error they introduce usually isn't detectable, and the final gathering process is substantially accelerated. *(Model courtesy of Florent Boyer.)*

Each time a photon is stored in `PhotonShootingTask::Run()`, a psuedo-random number is used to decide if a `RadiancePhoton` should be created for it; the fragment below takes care of doing so. The outgoing radiance values for the radiance photons can't be computed until all of the photons have been traced and the photon maps have been created. Therefore, the fragment here also stores the BSDF's reflectance and transmittance for use in computing the reflected radiance at this point later.

⟨*Possibly create radiance photon at photon intersection point*⟩ ≡ 810

```
if (depositedPhoton && integrator->finalGather &&
        rng.RandomFloat() < .125f) {
    Normal n = photonIsect.dg.nn;
    n = Faceforward(n, -photonRay.d);
    localRadiancePhotons.push_back(RadiancePhoton(photonIsect.dg.p, n));
    Spectrum rho_r = photonBSDF->rho(rng, BSDF_ALL_REFLECTION);
    localRpReflectances.push_back(rho_r);
    Spectrum rho_t = photonBSDF->rho(rng, BSDF_ALL_TRANSMISSION);
    localRpTransmittances.push_back(rho_t);
}
```

BSDF::rho() 482
BSDF_ALL_REFLECTION 428
BSDF_ALL_TRANSMISSION 428
DifferentialGeometry::nn 102
DifferentialGeometry::p 102
Faceforward() 66
Intersection::dg 186
Normal 65
PhotonIntegrator::finalGather 803
PhotonShootingTask::Run() 806
RadiancePhoton 815
Ray::d 67
RNG::RandomFloat() 1003
Spectrum 263

The RadiancePhoton structure just needs to store the position, surface normal, and outgoing radiance at the point. The surface normal is needed to distinguish between the two sides of the surface, since the outgoing radiance stored will only be used in the hemisphere around the normal. Typically the outgoing radiance on different sides of a surface will be quite different—distinguishing between the two hemispheres here reduces "light leak" image artifacts.

⟨*PhotonIntegrator Local Declarations*⟩ +≡
```
struct RadiancePhoton {
    RadiancePhoton(const Point &pp, const Normal &nn)
        : p(pp), n(nn), Lo(0.f) { }
    Point p;
    Normal n;
    Spectrum Lo;
};
```

The reflectances and transmittances only need to be stored temporarily during the preprocessing step between the time the RadiancePhoton is first created and the time that its outgoing radiance value is computed using the fully populated photon maps. Therefore, these values are stored in temporary arrays declared at the top of the Preprocess() photon shooting function.

⟨*Declare shared variables for photon shooting*⟩ +≡ 804
```
vector<Spectrum> rpReflectances, rpTransmittances;
```

Each PhotonShootingTask maintains a local array of these values; they are merged into the shared values after the mutex is acquired in the fragment ⟨*Merge local photon data with data in* PhotonIntegrator⟩.

⟨*Declare local variables for* PhotonShootingTask⟩ +≡ 806
```
vector<Spectrum> localRpReflectances, localRpTransmittances;
```

After the photon maps maps are all filled, exitant radiance can be computed at the radiance photons. The fragment below executes after the photon shooting tasks have finished and the kd-trees that store the photons have been created. It launches ComputeRadiance Tasks to compute the outgoing radiance at the radiance photons; when these tasks are done, it creates a kd-tree to store them.

⟨*Precompute radiance at a subset of the photons*⟩ ≡ 804
```
if (finalGather && radiancePhotons.size()) {
    ⟨Launch tasks to compute photon radiances⟩
    radianceMap = new KdTree<RadiancePhoton>(radiancePhotons);
}
```

⟨*PhotonIntegrator Private Data*⟩ +≡
```
KdTree<RadiancePhoton> *radianceMap;
```

Each task figures out what range of RadiancePhotons it's responsible for computing outgoing radiance values for and performs the computation for each one. The lookupBuf is temporary storage that will be passed to the photon interpolation routine.

KdTree 1029
Normal 65
PhotonIntegrator:: finalGather 803
PhotonIntegrator:: radianceMap 815
PhotonShootingTask 805
Point 63
RadiancePhoton 815
Spectrum 263

⟨*PhotonIntegrator Method Definitions*⟩ +≡
```
void ComputeRadianceTask::Run() {
    ⟨Compute range of radiance photons to process in task 816⟩
    ClosePhoton *lookupBuf = new ClosePhoton[nLookup];
    for (uint32_t i = rpStart; i < rpEnd; ++i) {
        ⟨Compute radiance for radiance photon i 816⟩
    }
    delete[] lookupBuf;
}
```

The radiance photons are partitioned among the tasks as equally as possible. Because the number of radiance photons and the number of tasks don't usually divide equally, the first `excess` tasks are given one more than `taskSize`. Computing the corresponding starting and ending photon for a particular task, then, must account for how many tasks prior to the current one had one extra.

⟨*Compute range of radiance photons to process in task*⟩ ≡ 816
```
uint32_t taskSize = radiancePhotons.size() / numTasks;
uint32_t excess = radiancePhotons.size() % numTasks;
uint32_t rpStart = min(taskNum, excess) * (taskSize+1) +
                   max(0, (int)taskNum-(int)excess) * taskSize;
uint32_t rpEnd = rpStart + taskSize + (taskNum < excess ? 1 : 0);
```

⟨*Compute radiance for radiance photon* i⟩ ≡ 816
```
RadiancePhoton &rp = radiancePhotons[i];
const Spectrum &rho_r = rpReflectances[i], &rho_t = rpTransmittances[i];
if (!rho_r.IsBlack()) {
    ⟨Accumulate outgoing radiance due to reflected irradiance 817⟩
}
if (!rho_t.IsBlack()) {
    ⟨Accumulate outgoing radiance due to transmitted irradiance⟩
}
```

To compute the reflected radiance at the photon, the product of the reflectance (stored when the `RadiancePhoton` was first created) and the irradiance (computed from the photon maps with the `EPhoton()` function) can be used to find an approximate outgoing radiance value for this point, ignoring directional variation in both the incident radiance distribution and the BSDF.

$$\begin{aligned} L_o(p, \omega_o) &= \int_{\mathcal{H}^2(\mathbf{n})} f_r(p, \omega_o, \omega_i)\, L_i(p, \omega_i)\, |\cos\theta_i|\, d\omega_i \\ &\approx \frac{1}{\pi}\rho_{hh}(p) \int_{\mathcal{H}^2(\mathbf{n})} L_i(p, \omega_i)\, |\cos\theta_i|\, d\omega_i \\ &= \frac{1}{\pi}\rho_{hh}(p)\, E(p, \mathbf{n}) \end{aligned}$$

ClosePhoton 822
EPhoton() 832
RadiancePhoton 815
Spectrum 263
Spectrum::IsBlack() 265

The `EPhoton()` utility function estimates the incident irradiance at the given point with the given surface normal using the photons in the photon map. It will be defined later

in this section, when we discuss how general measurements are computed using the photons.

⟨*Accumulate outgoing radiance due to reflected irradiance*⟩ ≡ 816

```
Spectrum E = EPhoton(directMap, nDirectPaths, nLookup, lookupBuf,
                     maxDistSquared, rp.p, rp.n) +
             EPhoton(indirectMap, nIndirectPaths, nLookup, lookupBuf,
                     maxDistSquared, rp.p, rp.n) +
             EPhoton(causticMap, nCausticPaths, nLookup, lookupBuf,
                     maxDistSquared, rp.p, rp.n);
rp.Lo += INV_PI * rho_r * E;
```

The process for computing outgoing radiance due to transmittance is similar and not included here.

15.6.4 USING THE PHOTON MAP

The photon mapping algorithm partitions the lighting integrand and uses different techniques to evaluate different parts of it. A number of different partitionings have been proposed, and different scenes with different illumination and reflection characteristics may benefit from others than the one implemented here. The `PhotonIntegrator` splits the BSDF into delta and nondelta components. For the nondelta BSDF components, incident radiance L_i is split into incident direct ($L_{i,d}$), caustic ($L_{i,c}$), and indirect ($L_{i,i}$) illumination:

$$\begin{aligned}
&\int_{S^2} f(p, \omega_o, \omega_i) L_i(p, \omega_i) |\cos\theta_i| \, d\omega_i \\
&\quad = \int_{S^2} f_\Delta(p, \omega_o, \omega_i) L_i(p, \omega_i) |\cos\theta_i| \, d\omega_i \\
&\qquad + \int_{S^2} f_{\neg\Delta}(p, \omega_o, \omega_i) L_{i,d}(p, \omega_i) |\cos\theta_i| \, d\omega_i \\
&\qquad + \int_{S^2} f_{\neg\Delta}(p, \omega_o, \omega_i) L_{i,c}(p, \omega_i) |\cos\theta_i| \, d\omega_i \\
&\qquad + \int_{S^2} f_{\neg\Delta}(p, \omega_o, \omega_i) L_{i,i}(p, \omega_i) |\cos\theta_i| \, d\omega_i.
\end{aligned} \tag{15.18}$$

The first term, delta BSDF components, is evaluated with regular recursive ray tracing, and the second term, direct lighting, is evaluated with the standard direct lighting algorithms of Section 15.1. The standard approaches to these two components handle them efficiently and accurately. The last two terms are evaluated using the photon maps.

We will elide the complete implementation of the `PhotonIntegrator::Li()` method, which is similar in form to the `Li()` methods of the other integrators and will focus on the fragments that evaluate the last two terms of the partitioning above, ⟨*Compute caustic lighting for photon map integrator*⟩ and ⟨*Compute indirect lighting for photon map integrator*⟩. When these fragments run, direct lighting and recursive ray tracing for the specular BSDF components have already been handled in the omitted code.

The `PhotonIntegrator` always uses the caustic photon map to account for light that has taken one or more specular bounces before hitting a nonspecular surface. These paths are particularly difficult to find when tracing paths starting from the point being

EPhoton() 832
INV_PI 1002
PhotonIntegrator 802
PhotonIntegrator::maxDistSquared 803
PhotonIntegrator::nLookup 803
RadiancePhoton::Lo 815
RadiancePhoton::n 815
RadiancePhoton::p 815
Spectrum 263

Figure 15.22: Caustics Cast by Light Focusing through a Complex Glass Object Have Remarkable Patterns. Photon mapping is particularly effective at capturing this effect.

shaded; fortunately, because caustics are focused light, there tend to be enough photons to compute good lighting estimates in areas with caustics. Figure 15.22 shows a caustic cast by the dragon model rendered with the photon map. It does a good job of accounting for this lighting effect.

The `LPhoton()` function, which will be described in Section 15.6.5, computes reflected radiance in the direction `wo` for the given BSDF and photon map. Using this function with the caustic photon map is all we need to include the effect of caustics in the computed radiance value here.

⟨*Compute caustic lighting for photon map integrator*⟩ ≡

```
ClosePhoton *lookupBuf = arena.Alloc<ClosePhoton>(nLookup);
L += LPhoton(causticMap, nCausticPaths, nLookup, lookupBuf, bsdf,
             rng, isect, wo, maxDistSquared);
```

Indirect illumination is usually handled with final gathering, though the indirect photon map can also be used directly via the `LPhoton()` function again.

⟨*Compute indirect lighting for photon map integrator*⟩ ≡

```
if (finalGather && indirectMap != NULL) {
    ⟨Do one-bounce final gather for photon map 819⟩
}
else
    L += LPhoton(indirectMap, nIndirectPaths, nLookup, lookupBuf,
                 bsdf, rng, isect, wo, maxDistSquared);
```

ClosePhoton 822
LPhoton() 831
PhotonIntegrator::
causticMap 813
PhotonIntegrator::
finalGather 803
PhotonIntegrator::
indirectMap 813
PhotonIntegrator::
maxDistSquared 803
PhotonIntegrator::
nCausticPaths 812
PhotonIntegrator::
nIndirectPaths 812
PhotonIntegrator::
nLookup 803

Final Gathering

When final gathering is being used to estimate reflected radiance to indirect illumination, we'd like to compute the value of the term

$$\int_{S^2} f_{\neg\Delta}(\mathrm{p}, \omega_\mathrm{o}, \omega_\mathrm{i})\, L_{\mathrm{i,i}}(\mathrm{p}, \omega_\mathrm{i})\, |\cos\theta_\mathrm{i}|\, \mathrm{d}\omega_\mathrm{i}$$

from Equation (15.18). As usual, in order to apply Monte Carlo integration for estimating this value, we need to sample incident directions ω_i from a distribution that we hope matches the shape of the integrand. The implementation here uses two sampling distributions: one based on the BSDF and one based on an approximation to the incident radiance function $L_{\mathrm{i,i}}$ that is constructed from the indirect illumination photons around the point p.

Given a sampled direction, the incident radiance $L_{\mathrm{i,i}}(\mathrm{p}, \omega_\mathrm{i})$ is found by tracing a ray to find the closest intersection and computing the outgoing radiance there using the single closest `RadiancePhoton`. Figure 15.23 illustrates the basic concept of final gathering, and Figure 15.24 shows the difference between using it or computing indirect lighting directly with the indirect lighting photons.

If the BSDF only has perfectly specular components, there's no reason to do final gathering. Otherwise, a collection of indirect lighting photons around the lookup point is found, and final gather rays are sampled.

⟨*Do one-bounce final gather for photon map*⟩ ≡ 818

```
BxDFType nonSpecular = BxDFType(BSDF_REFLECTION |
    BSDF_TRANSMISSION | BSDF_DIFFUSE | BSDF_GLOSSY);
if (bsdf->NumComponents(nonSpecular) > 0) {
    ⟨Find indirect photons around point for importance sampling 821⟩
    ⟨Copy photon directions to local array 823⟩
    ⟨Use BSDF to do final gathering 824⟩
    ⟨Use nearby photons to do final gathering 826⟩
}
```

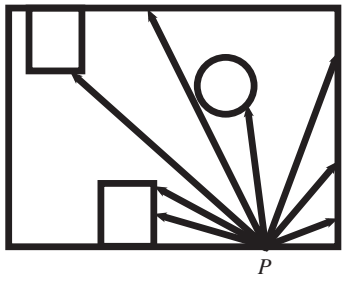


Figure 15.23: The final gathering step with photon mapping traces a set of rays to sample incident radiance at p in order to compute exitant radiance there. However, rather than continuing recursively and tracing more rays at the gather intersection points, the radiance photons are immediately used to compute the outgoing radiance there.

BSDF::NumComponents() 479
BSDF_DIFFUSE 428
BSDF_GLOSSY 428
BSDF_REFLECTION 428
BSDF_TRANSMISSION 428
BxDFType 428
RadiancePhoton 815

(a) (b) (c)

Figure 15.24: Scene (a) with direct illumination only, (b) with photon mapping but without final gathering, and (c) with final gathering. Note that the blurriness from using the photon map directly isn't a problem when final gathering is used. *(Model courtesy of Florent Boyer.)*

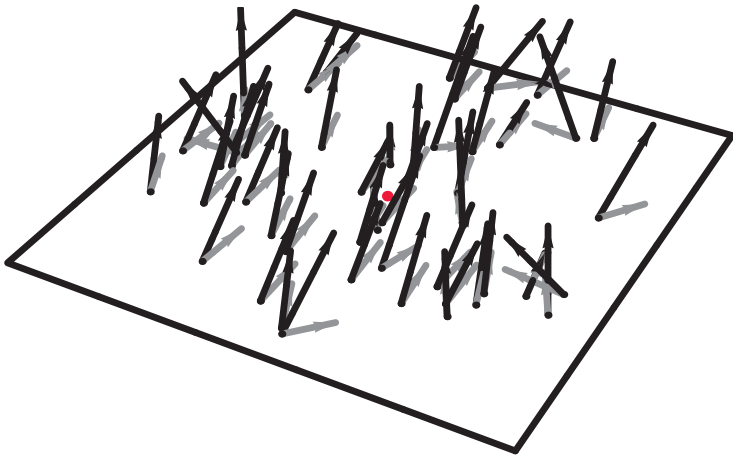


Figure 15.25: Incident directions of 50 photons around a point on the floor of the example scene. Many of the photons are arriving from a cluster of nearby directions. These directions can be used to define a distribution for importance sampling that approximates the distribution of indirect illumination at the point.

The photons around a point in the scene are not just useful for estimating the incident radiance distribution at the point in order to compute the outgoing reflected radiance: in their incident directions, they also carry useful information about the directional distribution of illumination at the point. This information can be put to use to define a sampling distribution for the final gathering rays that tries to match the distribution of indirect illumination. Figure 15.25 shows a plot of the incident directions of 50 photons around a point on the floor of the example scene; note that most of them have incident directions along a relatively similar set of directions.

When choosing ray directions for final gathering from a point, in some cases, sampling the BSDF will be the most effective strategy (for example, where the BSDF is very glossy and the amount of indirect illumination is similar in all directions); in others, sampling the indirect illumination will be a better approach (for example, when the indirect illumination is only from a small subset of directions, and the BSDF is diffuse). Allocating some samples to each approach works well in general, particularly when multiple importance sampling is used when evaluating the Monte Carlo estimator.

First, a fixed number of photons are searched for around the point being shaded by performing a lookup in the indirect photon map. The search radius is progressively widened until the desired number of photons is found. Because the squared distance parameter passed to the `KdTree::Lookup()` method may be modified during the lookup process, it's necessary to store the current search radius in a separate variable here, `searchDist2`.

⟨*Find indirect photons around point for importance sampling*⟩ ≡ 819
```
const uint32_t nIndirSamplePhotons = 50;
PhotonProcess proc(nIndirSamplePhotons,
                   arena.Alloc<ClosePhoton>(nIndirSamplePhotons));
float searchDist2 = maxDistSquared;
while (proc.nFound < nIndirSamplePhotons) {
    float md2 = searchDist2;
    proc.nFound = 0;
    indirectMap->Lookup(p, proc, md2);
    searchDist2 *= 2.f;
}
```

Like the octree used for the irradiance cache, the `KdTree` used to store photons here takes a parameter to its `Lookup()` method that provides an object with a particular method that is called for every item in the search region. `PhotonProcess` handles that role, storing information about the nearby photons that have been passed to it using the `ClosePhoton` structure.

⟨*PhotonIntegrator Local Declarations*⟩ +≡
```
struct PhotonProcess {
    ⟨PhotonProcess Public Methods⟩
    ClosePhoton *photons;
    uint32_t nLookup, nFound;
};
```

`ClosePhoton` 822
`KdTree::Lookup()` 1032
`MemoryArena::Alloc()` 1016
`PhotonIntegrator::maxDistSquared` 803
`PhotonProcess` 821
`PhotonProcess::nFound` 821

`ClosePhoton` represents a photon close to the lookup point. To keep track of a photon close to the lookup point, it is only necessary to store a pointer to it. However, the `ClosePhoton` structure also caches the squared distance from the photon to the lookup point in order to more quickly be able to discard the farthest away photon when a closer one is found.

⟨*PhotonIntegrator Local Declarations*⟩ +≡
```
struct ClosePhoton {
    ⟨ClosePhoton Public Methods 822⟩
    const Photon *photon;
    float distanceSquared;
};
```

⟨*ClosePhoton Public Methods*⟩ ≡ 822
```
ClosePhoton(const Photon *p = NULL, float md2 = INFINITY)
    : photon(p), distanceSquared(md2) { }
```

`ClosePhoton` also provides an ordering relation, so that the farthest away photon in a collection of photons can be found.

⟨*ClosePhoton Public Methods*⟩ +≡ 822
```
bool operator<(const ClosePhoton &p2) const {
    return distanceSquared == p2.distanceSquared ?
        (photon < p2.photon) : (distanceSquared < p2.distanceSquared);
}
```

As the `KdTree` traverses the tree nodes, it calls the `operator()` method of `PhotonProcess` for each photon inside the search radius. This method stores a pointer to each such photon with the `ClosePhotons` array, `photons`.

⟨*PhotonIntegrator Local Definitions*⟩ ≡
```
inline void PhotonProcess::operator()(const Point &p,
        const Photon &photon, float distSquared, float &maxDistSquared) {
    if (nFound < nLookup) {
        ⟨Add photon to unordered array of photons 823⟩
    }
    else {
        ⟨Remove most distant photon from heap and add new photon 823⟩
    }
}
```

Until `nLookup` photons have been found around the point, this method stores the nearby photons in an unordered array that is filled in whatever order photons are passed to this callback method. However, once the `nLookup+1`st photon arrives (if it does), it is necessary to discard the photon that is farthest away and keep only the `nLookup` closest ones. Therefore, at that point, the `photons` array is reordered to be a heap, such that the root element is the one with the greatest distance from the lookup point. Recall that a heap data structure can be constructed in linear time and that it can be updated after an item is removed or added in logarithmic time. It is much more efficient to organize photons with a heap than to keep them sorted by distance. The `make_heap()` function in the C++ standard library reorders a given array into a heap so that the zeroth element is the root of the heap.

Furthermore, once `nLookup` photons have been found, the search radius that the `KdTree` uses as it traverses its nodes can be decreased; there's no reason to consider any additional

ClosePhoton 822
ClosePhoton::distanceSquared 822
ClosePhoton::photon 822
INFINITY 1002
Photon 805
PhotonProcess 821
PhotonProcess::nFound 821
PhotonProcess::nLookup 821
Point 63

photons that are farther away than the farthest one in the heap. `KdTree::Lookup()` passes a reference to `maxDistSquared` into this callback method so that the callback method can reduce its value in situations like this.

⟨*Add photon to unordered array of photons*⟩ ≡ 822
```
photons[nFound++] = ClosePhoton(&photon, distSquared);
if (nFound == nLookup) {
    std::make_heap(&photons[0], &photons[nLookup]);
    maxDistSquared = photons[0].distanceSquared;
}
```

As additional photons come in after the heap has been built, we know that the squared distance to new ones must be less than the `maxDistSquared`, since the `KdTree` doesn't call the callback method for items that are farther away. Thus, any additional photon must be closer than the root node of the heap, which is the farthest away one that is stored. Therefore, the method immediately calls the standard library routine that removes the root item of the heap and updates the order of the remaining ones in the heap to reestablish a valid heap, `pop_heap()`. The new photon is added to the end of the array, which will have been left empty after `pop_heap()` was called and then `push_heap()` again rebuilds the heap, accounting for a new element added to the end of it. After all of this, `maxDistSquared` can again be reduced to whatever the distance is to the new farthest away photon.

⟨*Remove most distant photon from heap and add new photon*⟩ ≡ 822
```
std::pop_heap(&photons[0], &photons[nLookup]);
photons[nLookup-1] = ClosePhoton(&photon, distSquared);
std::push_heap(&photons[0], &photons[nLookup]);
maxDistSquared = photons[0].distanceSquared;
```

Once the desired number of photons has been found, their directions are copied to a contiguous array. This extra step improves performance for the rest of the final gathering work by ensuring that all of the photons' directions can be found without incurring excessive cache misses. If instead the original `Photon` structures pointed to by `proc.photons[].photon` were used, the pattern of memory accesses would be relatively incoherent and unused extra data from the `Photon` structure would potentially be repeatedly loaded into the cache.

⟨*Copy photon directions to local array*⟩ ≡ 819
```
Vector *photonDirs = arena.Alloc<Vector>(nIndirSamplePhotons);
for (uint32_t i = 0; i < nIndirSamplePhotons; ++i)
    photonDirs[i] = proc.photons[i].photon->wi;
```

Once the incident directions of the indirect photons around the point are available, the final gathering process can begin. In the first stage, samples are taken from the BSDF to determine outgoing directions.

ClosePhoton 822
ClosePhoton::
distanceSquared 822
MemoryArena::Alloc() 1016
Photon 805
Photon::wi 805
PhotonProcess::nFound 821
PhotonProcess::nLookup 821
PhotonProcess::photons 821
Vector 57

⟨*Use BSDF to do final gathering*⟩ ≡ 819
```
Spectrum Li = 0.;
for (int i = 0; i < gatherSamples; ++i) {
    ⟨Sample random direction from BSDF for final gather ray 824⟩
    ⟨Trace BSDF final gather ray and accumulate radiance 824⟩
}
L += Li / gatherSamples;
```

Specular components of the BSDF are ignored when choosing these samples, since they are handled separately with recursive ray tracing.

⟨*Sample random direction from BSDF for final gather ray*⟩ ≡ 824
```
Vector wi;
float pdf;
BSDFSample bsdfSample(sample, bsdfGatherSampleOffsets, i);
Spectrum fr = bsdf->Sample_f(wo, &wi, bsdfSample,
                             &pdf, BxDFType(BSDF_ALL & ~BSDF_SPECULAR));
if (fr.IsBlack() || pdf == 0.f) continue;
```

If an intersection is found for the final gather ray, then the outgoing radiance at the intersection is found using the radiance photons, and a weight for the sample is computed with multiple importance sampling. Since both the BSDF and the indirect lighting distribution approximated by the nearby photons may match important components of the integrand, MIS improves the quality of the final results in cases where one or the other of the distributions matches the integrand much better than the other one.

⟨*Trace BSDF final gather ray and accumulate radiance*⟩ ≡ 824
```
RayDifferential bounceRay(p, wi, ray, isect.rayEpsilon);
Intersection gatherIsect;
if (scene->Intersect(bounceRay, &gatherIsect)) {
    ⟨Compute exitant radiance Lindir using radiance photons 824⟩
    ⟨Compute MIS weight for BSDF-sampled gather ray 825⟩
    Li += fr * Lindir * (AbsDot(wi, n) * wt / pdf);
}
```

Given a lookup point and its surface normal, an instance of the `RadiancePhotonProcess` structure is passed to the `KdTree::Lookup()` method. It stores a pointer to the single closest `RadiancePhoton` to the lookup point. Its contribution is then attenuated by participating media, if any.

⟨*Compute exitant radiance* `Lindir` *using radiance photons*⟩ ≡ 824, 827
```
Spectrum Lindir = 0.f;
Normal nGather = gatherIsect.dg.nn;
nGather = Faceforward(nGather, -bounceRay.d);
RadiancePhotonProcess proc(nGather);
float md2 = INFINITY;
radianceMap->Lookup(gatherIsect.dg.p, proc, md2);
if (proc.photon != NULL)
    Lindir = proc.photon->Lo;
Lindir *= renderer->Transmittance(scene, bounceRay, NULL, rng, arena);
```

AbsDot() 61
BSDF::Sample_f() 706
BSDFSample 705
BSDF_ALL 428
BSDF_SPECULAR 428
BxDFType 428
DifferentialGeometry::nn 102
DifferentialGeometry::p 102
Faceforward() 66
INFINITY 1002
Intersection 186
Intersection::dg 186
Intersection::rayEpsilon 186
KdTree::Lookup() 1032
Normal 65
PhotonIntegrator::
bsdfGatherSampleOffsets 803
RadiancePhoton 815
RadiancePhoton::Lo 815
RadiancePhotonProcess 825
RadiancePhotonProcess::
photon 825
Ray::d 67
RayDifferential 69
Renderer::Transmittance() 25
Scene::Intersect() 23
Spectrum 263
Spectrum::IsBlack() 265
Vector 57

The RadiancePhotonProcess structure only needs to record the single closest radiance photon to the lookup point, so the photon pointer suffices for this. Because radiance photons only store illumination in a hemisphere around their surface normal, it also needs to store the surface normal to ensure that the surface normal here and the normal of radiance photon used are in the same hemisphere.

⟨*PhotonIntegrator Local Declarations*⟩ +≡
```
struct RadiancePhotonProcess {
    ⟨RadiancePhotonProcess Methods 825⟩
    const Normal &n;
    const RadiancePhoton *photon;
};
```

Along the lines of PhotonProcess, operator() of the RadiancePhotonProcess structure is called by the kd-traversal method for each photon within the search radius. Because it is only necessary to record the single closest photon to the lookup point, the implementation here is particularly straightforward. In looks for the single closest radiance photon with normal pointing in the same hemisphere as the normal at the point the final gather ray intersected a surface, storing a pointer to it and reducing the maximum search radius when it finds one. (Recall that KdTree::Lookup() doesn't call this method for any items outside the search radius, so it's not necessary to check that the given RadiancePhoton is the closest one here.)

⟨*RadiancePhotonProcess Methods*⟩ ≡ 825
```
void operator()(const Point &p, const RadiancePhoton &rp,
                float distSquared, float &maxDistSquared) {
    if (Dot(rp.n, n) > 0) {
        photon = &rp;
        maxDistSquared = distSquared;
    }
}
```

To compute the multiple importance sampling weight, we need to compute the value of the PDF for sampling the direction wi from the photons' distribution. The fragment that does this, ⟨*Compute PDF for photon-sampling of direction* wi⟩, will be defined shortly, after that sampling approach used is explained.

⟨*Compute MIS weight for BSDF-sampled gather ray*⟩ ≡ 824
```
⟨Compute PDF for photon-sampling of direction wi 827⟩
float wt = PowerHeuristic(gatherSamples, pdf, gatherSamples, photonPdf);
```

In an effort to generate samples according to the incident radiance distribution, the sampling approach here takes the photons around the point being shaded and defines a cone of directions centered about each photon's incident direction. The sampling method then chooses one of the photons at random and randomly samples a direction inside its cone to select a final gather ray.

Note that this sampling approach may well never generate samples for some directions ω_i where the amount of indirect illumination is nonzero. Therefore, this wouldn't be an acceptable sampling technique if it was the only method being used with the standard

Normal 65
PhotonIntegrator::
gatherSamples 803
PhotonProcess 821
Point 63
PowerHeuristic() 693
RadiancePhoton 815

Monte Carlo estimator, since there must be some probability of generating any sample with nonzero contribution. However, when using multiple importance sampling, it is only necessary that *one* of the sampling methods used have nonzero probability of sampling directions where the integrand is nonzero. Because this property is required for the BSDF sampling methods, this shortcoming of the photon-based sampling method we have chosen isn't a problem here.

⟨*Use nearby photons to do final gathering*⟩ ≡ 819
```
Li = 0.;
for (int i = 0; i < gatherSamples; ++i) {
    ⟨Sample random direction using photons for final gather ray 826⟩
    ⟨Trace photon-sampled final gather ray and accumulate radiance 827⟩
}
L += Li / gatherSamples;
```

This method first chooses a photon with uniform probability from the collection of nearby photons. (If photon weights were expected to vary significantly, we might want to instead construct a photon sampling distribution based on the photon weights.) Here, we are slightly abusing the semantics of the `BSDFSample` type, using the component sample to choose which photon to use.

⟨*Sample random direction using photons for final gather ray*⟩ ≡ 826
```
BSDFSample gatherSample(sample, indirGatherSampleOffsets, i);
int photonNum = min((int)nIndirSamplePhotons - 1,
    Floor2Int(gatherSample.uComponent * nIndirSamplePhotons));
⟨Sample gather ray direction from photonNum 826⟩
```

Given the photon, a direction is sampled from a cone about its incident direction with uniform probability. This is easily done with the `UniformSampleCone()` routine. Again we are overloading the `BSDFSample` semantics, using its 2D directional sample to sample the direction in the cone. The member variable `cosGatherAngle` is initialized in the `PhotonIntegrator` constructor based on a parameter that can be set in the input file. By default, rays are distributed in a cone that subtends an angle of 10 degrees.

⟨*Sample gather ray direction from* `photonNum`⟩ ≡ 826
```
Vector vx, vy;
CoordinateSystem(photonDirs[photonNum], &vx, &vy);
Vector wi = UniformSampleCone(gatherSample.uDir[0], gatherSample.uDir[1],
                              cosGatherAngle, vx, vy, photonDirs[photonNum]);
```

The same process is used for computing the contributions of final gather rays as was used for rays found by sampling the BSDF; therefore, we reuse the fragment ⟨*Compute exitant radiance* `Lindir` *using radiance photons*⟩ here.

BSDFSample 705
BSDFSample::uComponent 705
BSDFSample::uDir 705
Floor2Int() 1002
PhotonIntegrator::
cosGatherAngle 803
PhotonIntegrator::
indirGatherSampleOffsets 803
UniformSampleCone() 713
Vector 57

⟨*Trace photon-sampled final gather ray and accumulate radiance*⟩ ≡ 826
```
Spectrum fr = bsdf->f(wo, wi);
if (fr.IsBlack()) continue;
RayDifferential bounceRay(p, wi, ray, isect.rayEpsilon);
Intersection gatherIsect;
if (scene->Intersect(bounceRay, &gatherIsect)) {
    ⟨Compute exitant radiance Lindir using radiance photons 824⟩
    ⟨Compute PDF for photon-sampling of direction wi 827⟩
    ⟨Compute MIS weight for photon-sampled gather ray 827⟩
    Li += fr * Lindir * AbsDot(wi, n) * wt / photonPdf;
}
```

It's easy to compute the PDF for sampling a given direction with the nearby photons: it's the average of the PDFs for each of the photons to sample the direction. For each photon, a dot product indicates if the given direction is within the cone of possible directions for that photon; if the direction is inside the cone, the constant PDF for sampling a direction in a cone of the given angle is added to the PDF sum.

A small fudge factor is used in the test for whether a given direction is within a given cone in order to handle the case where `UniformSampleCone()` selected a direction at the very edge of the cone. In that case, the dot product test here might indicate that the direction wasn't in the cone it was sampled from, leading to `photonPdf` being zero and thence to not-a-number or infinite values in the final image.

⟨*Compute PDF for photon-sampling of direction* wi⟩ ≡ 825, 827
```
float photonPdf = 0.f;
float conePdf = UniformConePdf(cosGatherAngle);
for (uint32_t j = 0; j < nIndirSamplePhotons; ++j)
    if (Dot(photonDirs[j], wi) > .999f * cosGatherAngle)
        photonPdf += conePdf;
photonPdf /= nIndirSamplePhotons;
```

To find the MIS weight for the direction sampled using the incident photons, the BSDF's PDF for choosing the direction must be computed.

⟨*Compute MIS weight for photon-sampled gather ray*⟩ ≡ 827
```
float bsdfPdf = bsdf->Pdf(wo, wi);
float wt = PowerHeuristic(gatherSamples, photonPdf, gatherSamples, bsdfPdf);
```

AbsDot() 61
BSDF::f() 481
BSDF::Pdf() 708
Dot() 60
Intersection 186
LPhoton() 831
PhotonIntegrator::cosGatherAngle 803
PhotonIntegrator::gatherSamples 803
PowerHeuristic() 693
RayDifferential 69
Scene::Intersect() 23
Spectrum 263
Spectrum::IsBlack() 265
UniformConePdf() 713

15.6.5 PHOTON INTERPOLATION AND DENSITY ESTIMATION

We will now introduce two functions that compute measurements using the photons in the photon maps and connect their implementations to the particle tracing theory introduced in Section 15.6.1. The first function, `LPhoton()`, computes the reflected radiance at a point with a given BSDF due to incident illumination. The second, `EPhoton()`, computes the incident irradiance at a point.

In order to compute reflected radiance at a point, we need to estimate the exitant radiance equation at a point p in a direction ω_o, which can equivalently (and cumbersomely) be

written as a measurement over all points on surfaces in the scene where a Dirac delta distribution selects only particles precisely at p:

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

and so the function that describes the measurement is

$$W_{\mathrm{e}}(\mathrm{p}', \omega) = \delta(\mathrm{p}' - \mathrm{p})\ f(\mathrm{p}, \omega_{\mathrm{o}}, \omega)\ |\cos\theta|.$$

Unfortunately, because there is a delta distribution in W_{e}, all of the particle histories that were generated during the particle-tracing step have zero probability of having nonzero contribution if Equation (15.15) is used to compute the estimate of the measurement value (just as we will never be able to choose a direction from a diffuse surface that intersects a point light source unless the direction is sampled accounting for this).

Here is the point at which bias is introduced into the photon mapping algorithm. Under the assumption that the information about illumination at nearby points can be used to construct an estimate of illumination at the shading point, photon mapping interpolates information about illumination at the point being shaded from nearby photons. The more photons there are around the point and the higher their weights, the more radiance is estimated to be incident at the point. (Recall from the original description of the particle-tracing algorithm and Equation (15.15) that both the local density of the particles and their individual weights together affect their contribution.) This estimated radiance is used in conjunction with the surface's BSDF to compute the reflected light.

A statistical technique called *density estimation* provides the mathematical tools to perform this interpolation. Density estimation constructs a PDF given a set of sample points under the assumption that the samples are distributed according to the overall distribution of some function of interest. Histogramming is a straightforward example of the idea. In 1D, the line is divided into intervals with some width, and one can count how many samples land in each interval and normalize so that the areas of the intervals sum to one.

Kernel methods are a more sophisticated density estimation technique. They generally give better results and smoother PDFs that don't suffer from the discontinuities that histograms do. Given a kernel function $k(x)$ that integrates to one,

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

the kernel estimator for N samples at locations x_i is

$$\hat{p}(x) = \frac{1}{Nh} \sum_{i=1}^{N} k\left(\frac{x - x_i}{h}\right),$$

where h is the window width (also known as the smoothing parameter). Kernel methods can be thought of as placing a series of bumps at observation points, where the sum of the

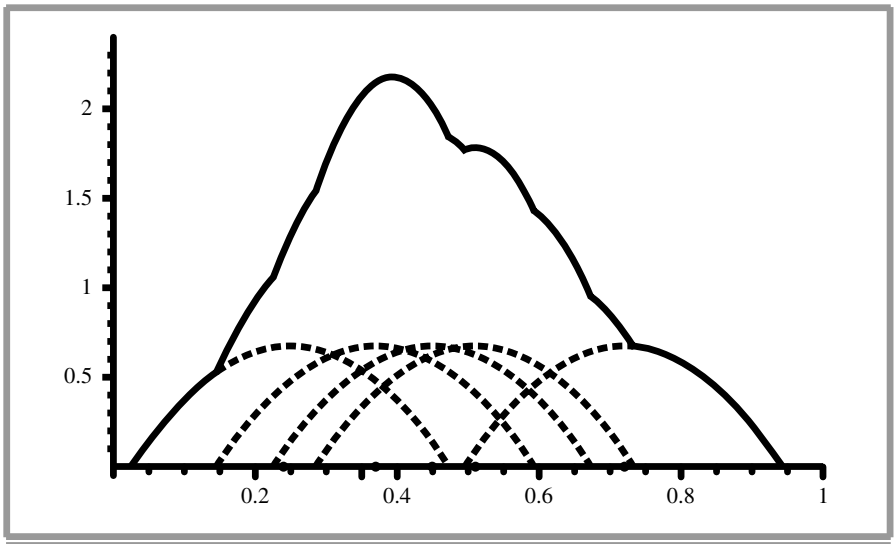


Figure 15.26: 1D example of density estimation, using the Epanechnikov kernel, $k(t) = .75(1 - .2t^2)/\sqrt{5}$, if $t < \sqrt{5}$, 0 otherwise, and a width of 0.1. The points marked with closed circles are the sample points, and an instance of the kernel (dashed lines) is placed over each one. The sum of the kernels gives a properly normalized PDF that attempts to model a distribution that the points could be distributed by.

bumps forms a PDF since they individually integrate to one and the sum is normalized. Figure 15.26 shows an example of density estimation in 1D, where a smooth PDF is computed from a set of sample points.

The key question with kernel methods is how the window width h is chosen. If it is too wide, the PDF will blur out relevant detail in parts of the domain with many samples; if it is too narrow, the PDF will be too bumpy in the tails of the distribution where there aren't many samples. Nearest-neighbor techniques solve this problem by choosing h adaptively based on local density of samples. Where there are many samples, the width is small; where there are few samples, the width is large. For example, one approach is to pick a number n and find the distance to the nth nearest sample from the point x and use that distance, $d_n(x)$, for the window width. This is the *generalized nth nearest-neighbor estimate:*

$$\hat{p}(x) = \frac{1}{N d_n(x)} \sum_{i=1}^{N} k\left(\frac{x - x_i}{d_n(x)}\right).$$

In d dimensions, this generalizes to

$$\hat{p}(x) = \frac{1}{N (d_n(x))^d} \sum_{i=1}^{N} k\left(\frac{x - x_i}{d_n(x)}\right). \qquad [15.19]$$

The implementation here uses Simpson's kernel,

$$k(x) = \begin{cases} \frac{3}{\pi}\left(1 - \left(\frac{d_i(x)}{d_n(x)}\right)^2\right)^2 & |x| < 1 \\ 0 & \text{otherwise.} \end{cases}$$

where $d_i(x)$ is the distance to the ith point used in the density estimate.

Intuitively, it makes sense to use a kernel that gives greater weight to samples near the lookup point and less weight to samples farther away. Doing so is generally a good idea, since the far-away samples give less useful information about the function than the nearby ones. Using this kernel for photon mapping thus also smoothes out the illumination estimates computed by the photon map, a generally desirable property (sharp caustics aside).

The implementation here assumes that no photons with squared distance from the point greater than `maxDist2` will be passed to the kernel function.

⟨*PhotonIntegrator Local Definitions*⟩ +≡

```
inline float kernel(const Photon *photon, const Point &p,
                    float maxDist2) {
    float s = (1.f - DistanceSquared(photon->p, p) / maxDist2);
    return 3.f * INV_PI * s * s;
}
```

Substituting into the measurement equation, it can be shown that the appropriate estimator for the measurement we'd like to compute, the exitant radiance at the point p in direction ω, is given by

$$L_o(\mathrm{p}, \omega_o) \approx \sum_j^N \hat{p}(\mathrm{p}_j)\, \alpha_j\, f(\mathrm{p}, \omega_o, \omega_j), \qquad [15.20]$$

where the sum is over all of the photons and scale factors for the photons are computed based on the density estimation, Equation (15.19). Because we know that the kernel function is zero for points farther away than the nth nearest neighbor distance $d_n(x)$, implementations of this sum only need to sum over the n closest neighbors.

The error introduced by this interpolation can be difficult to quantify. Storing more photons, so that it isn't necessary to use photons as far away, will almost always improve the results, but in general, error will depend on how quickly the illumination is changing at the point. One can always construct pathological cases where this error is unacceptable, but in practice it usually isn't too bad. Because the interpolation step tends to blur out illumination, high-frequency changes in lighting are sometimes poorly reconstructed with photon mapping. Since indirect illumination tends to be low frequency, this isn't too much of a problem in practice. And, although caustics do often have high-frequency variation, photon mapping also works well for them since many photons are focused in caustic regions.

Given this background, the `LPhoton()` function has two main tasks: First, it needs to find the `nLookup` nearest photons to the point where the shading is being done. Second, it needs to use those photons to compute the reflected radiance at the point. The function

DistanceSquared() 65
INV_PI 1002
LPhoton() 831
Photon 805
Photon::p 805
Point 63

starts by checking if the BSDF has any nonspecular components—if it is purely specular, there's no need to do the lookup, since the BSDF won't return any contribution for any of the photons' incident directions.

```
⟨PhotonIntegrator Local Definitions⟩ +≡
  Spectrum LPhoton(KdTree<Photon> *map, int nPaths, int nLookup,
        ClosePhoton *lookupBuf, BSDF *bsdf, RNG &rng,
        const Intersection &isect, const Vector &wo, float maxDist2) {
      Spectrum L(0.);
      BxDFType nonSpecular = BxDFType(BSDF_REFLECTION |
          BSDF_TRANSMISSION | BSDF_DIFFUSE | BSDF_GLOSSY);
      if (map && bsdf->NumComponents(nonSpecular) > 0) {
          ⟨Do photon map lookup at intersection point 831⟩
          ⟨Estimate reflected radiance due to incident photons 831⟩
      }
      return L;
  }
```

The previously introduced `PhotonProcess` object and the `KdTree` take care of finding the `nLookup` nearest neighbors.

```
⟨Do photon map lookup at intersection point⟩ ≡ 831
  PhotonProcess proc(nLookup, lookupBuf);
  map->Lookup(isect.dg.p, proc, maxDist2);
```

Once the photons have been found, the value of Equation (15.20) can be computed. If the surface is purely diffuse, a separate code path is used, since those surfaces can be handled more efficiently.

```
⟨Estimate reflected radiance due to incident photons⟩ ≡ 831
  ClosePhoton *photons = proc.photons;
  int nFound = proc.nFound;
  Normal Nf = Faceforward(bsdf->dgShading.nn, wo);
  if (bsdf->NumComponents(BxDFType(BSDF_REFLECTION |
          BSDF_TRANSMISSION | BSDF_GLOSSY)) > 0) {
      ⟨Compute exitant radiance from photons for glossy surface 832⟩
  }
  else {
      ⟨Compute exitant radiance from photons for diffuse surface⟩
  }
```

If the BSDF has any nondiffuse components, `LPhoton()` loops over the nearby photons to evaluate the sum in Equation (15.20).

BSDF 478
BSDF_DIFFUSE 428
BSDF_GLOSSY 428
BSDF_REFLECTION 428
BSDF_TRANSMISSION 428
BxDFType 428
ClosePhoton 822
ClosePhoton::photon 822
DifferentialGeometry::p 102
Faceforward() 66
Intersection 186
Intersection::dg 186
KdTree 1029
KdTree::Lookup() 1032
Normal 65
Photon 805
PhotonIntegrator::
  nLookup 803
PhotonProcess 821
RNG 1003
Spectrum 263
Vector 57

⟨*Compute exitant radiance from photons for glossy surface*⟩ ≡ 831
```
for (int i = 0; i < nFound; ++i) {
    const Photon *p = photons[i].photon;
    float k = kernel(p, isect.dg.p, maxDist2);
    L += (k / (nPaths * maxDist2)) * bsdf->f(wo, p->wi) *
         p->alpha;
}
```

For purely diffuse BSDFs, it's wasteful to call the `BSDF::f()` method `nFound` times, since it will always return a constant value. In this case, the equivalent computation can be done much more efficiently by finding the weighted sum of photon α values and multiplying it by the constant BSDF, found by dividing the surface's reflectance by π. We won't include the fragment that does this, ⟨*Compute exitant radiance from photons for diffuse surface*⟩, since its implementation is straightforward.

Following a similar approach, the `EPhoton()` function estimates incident irradiance at a given point about the hemisphere with a given normal using the provided photon map.

⟨*PhotonIntegrator Local Definitions*⟩ +≡
```
Spectrum EPhoton(KdTree<Photon> *map, int count, int nLookup,
        ClosePhoton *lookupBuf, float maxDist2, const Point &p,
        const Normal &n) {
    if (!map) return 0.f;
    ⟨Lookup nearby photons at irradiance computation point 832⟩
    ⟨Accumulate irradiance value from nearby photons 833⟩
}
```

This method first performs the usual photon map lookup, finding the given number of photons around the lookup point.

⟨*Lookup nearby photons at irradiance computation point*⟩ ≡ 832
```
PhotonProcess proc(nLookup, lookupBuf);
float md2 = maxDist2;
map->Lookup(p, proc, md2);
```

Given these photons, density estimation is again performed, this time to estimate the incident irradiance at the point. How to compute the value of this estimate can be determined by defining a measurement function to compute irradiance at a point p with normal **n**:

$$W_e(\mathrm{p}', \omega) = \delta(\mathrm{p}' - \mathrm{p}) \begin{cases} 1 & (\mathbf{n} \cdot \omega) > 0 \\ 0 & \text{otherwise,} \end{cases}$$

and following the same approach used for `LPhoton()` to derive how to use photons to compute reflected radiance using density estimation. Note that photons in the opposite hemisphere from the given surface normal are ignored, as they don't contribute to the outgoing radiance in the hemisphere of interest. Here, also, a constant kernel is used since the irradiance values are only used for the radiance photons, which aren't seen directly.

BSDF::f() 481
ClosePhoton 822
ClosePhoton::photon 822
DifferentialGeometry::p 102
EPhoton() 832
Intersection::dg 186
KdTree 1029
KdTree::Lookup() 1032
LPhoton() 831
Normal 65
Photon 805
Photon::alpha 805
PhotonProcess 821
Point 63
Spectrum 263

⟨*Accumulate irradiance value from nearby photons*⟩ ≡ 832

```
if (proc.nFound == 0) return Spectrum(0.f);
ClosePhoton *photons = proc.photons;
Spectrum E(0.);
for (uint32_t i = 0; i < proc.nFound; ++i)
    if (Dot(n, photons[i].photon->wi) > 0.)
        E += photons[i].photon->alpha;
return E / (count * md2 * M_PI);
```

*15.7 METROPOLIS LIGHT TRANSPORT

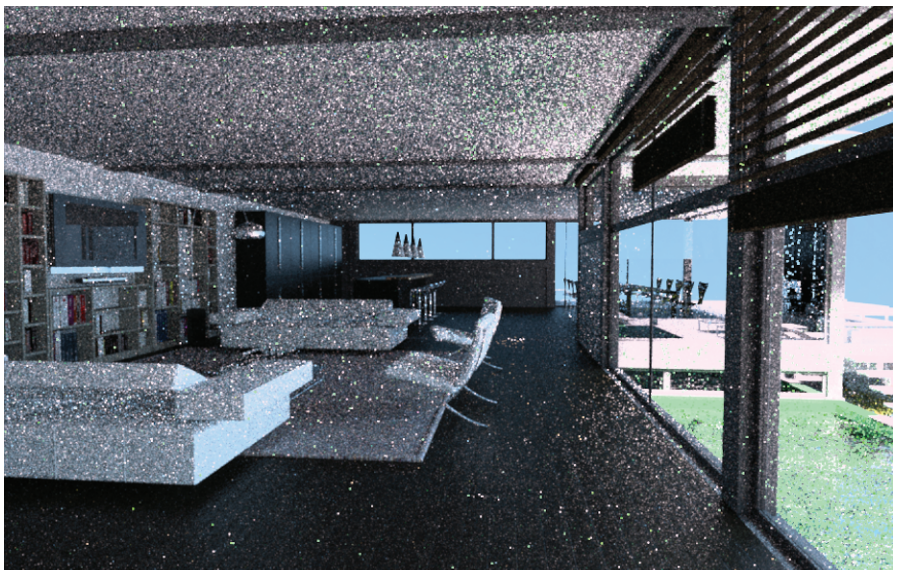
This chapter concludes with a `Renderer` that uses Metropolis sampling (first introduced in Section 13.4) as the foundation of an unbiased method for rendering images, Metropolis Light Transport (MLT).

MLT generates a sequence of light-carrying paths through the scene, where each path is found by mutating the previous path in some manner. These mutations are done in a way that ensures that the overall distribution of sampled paths is proportional to the contribution the paths make to the image being generated. Such a distribution of paths can in turn be used to generate an image of the scene. Given the flexibility of the Metropolis sampling method, there are relatively few restrictions on the types of mutations that can be applied; in general, it is possible to apply unusual sampling techniques designed to sample otherwise difficult-to-find light-carrying paths. Many such sampling techniques that can be used with MLT can't easily be applied to other Monte Carlo light transport algorithms without introducing bias.

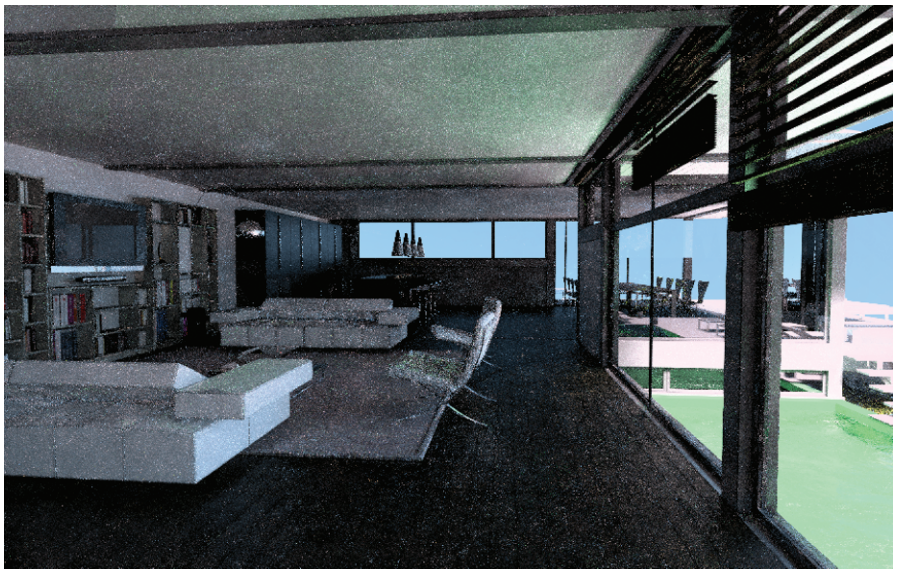
MLT has another important advantage with respect to previous unbiased approaches for image synthesis in that it performs *local exploration*: when a path that makes a large contribution to the image is found, it's easy to sample other paths that are similar to that one by making small perturbations to it. When a function has a small value over most of its domain and a large contribution in only a small subset of it, local exploration amortizes the expense (in samples) of the search for the important region by taking a number of samples from this part of the path space. This property makes MLT a reasonably robust light transport algorithm: while it is about as efficient as other unbiased techniques (e.g., path tracing or bidirectional ray tracing) for relatively straightforward lighting problems, it distinguishes itself in more difficult settings where most of the light transport happens along a small fraction of all of the possible paths through the scene.

ClosePhoton 822
ClosePhoton::photon 822
M_PI 1002
Photon::alpha 805
Photon::wi 805
PhotonProcess::nFound 821
PhotonProcess::photons 821
Renderer 24
Spectrum 263

Figure 15.27 shows the contemporary house scene rendered with path tracing and MLT, with an equal number of samples taken for each approach, and Figure 15.28 compares them with the San Miguel scene. For both, MLT generates a better result, but the difference is particularly pronounced with the house scene, which is a particularly challenging scene for light transport algorithms in that there is essentially no direct illumination inside the house; all light-carrying paths must follow specular bounces through the glass windows. Table 15.1 helps illustrate why this is so: for both of these scenes, regular path tracing has a lot of trouble finding paths that carry any radiance, while Metropolis is more effective at doing so thanks to path reuse.

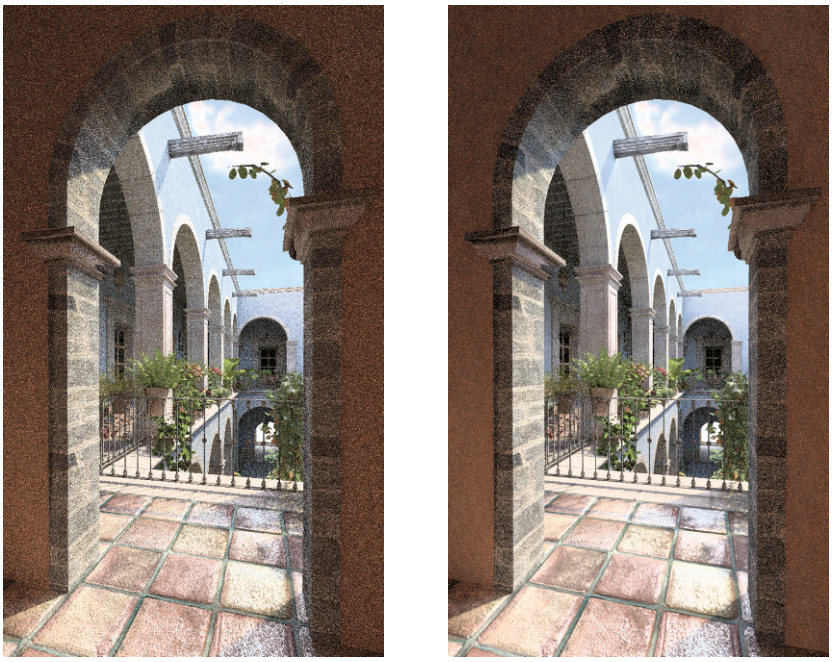


(a)



(b)

Figure 15.27: Comparison of Path Tracing and Metropolis Light Transport. (a) Rendered with path tracing with 256 samples per pixel. Even with many samples, the image is quite noisy. (b) Rendered with Metropolis Light Transport with an average of 256 mutations per pixel. MLT generates a substantially better image for the same amount of work. Path tracing does do better in a few localized parts of the image: note the back wall underneath the windows, the bevel around the television, and the wooden blinds by the window on the upper right, for example. Each of these is a relatively dark region of the image; the fixed sampling rate of the path tracing integrator gives them more samples than Metropolis, which adapts the sampling rate to do more work in brighter areas. *(Model courtesy of Florent Boyer.)*



(a) (b)

Figure 15.28: Comparison of Path Tracing and Metropolis Light Transport. (a) Rendered with path tracing with 256 samples per pixel. (b) Rendered with Metropolis Light Transport with an average of 256 mutations per pixel. As with Figure 15.27, MLT is more effective, but, because this is a less difficult scene in which to find light transport paths, the difference is less striking here. *(Model courtesy of Guillermo M. Leal Llaguno.)*

Table 15.1: Percentage of Traced Paths That Carried Zero Radiance. With these two scenes, path tracing has trouble finding light-carrying paths: only one in 10 or one in 20 ray paths traced carries any radiance at all. Thanks to local exploration, Metropolis is better able to find additional light-carrying paths after one has been found. This is one of the reasons why it is more efficient than path tracing here.

	Path Tracing	Metropolis
Modern House	95.8%	53.6%
San Miguel	89.4%	50.6%

In order to see how to apply Metropolis to the light transport problem, recall the image function formalism introduced in Section 7.1.5. First, consider the case of rendering a scene with Whitted ray tracing and no area light sources: in this case, integration at a surface is deterministic and requires no random sampling, so the contribution of a ray is fully determined by its 5D camera sample of image position, time, and lens position. In this case, the image contribution function is $L(x, y, t, u, v)$, where x and y are the image position, t is the ray's time, and u and v give the lens position.

In the more general case of path tracing, L is an infinite-dimensional function $L(X_1, X_2, \ldots)$, where all of the random samples used in integration are represented by elements of X_i.[8] Beyond the camera samples, the additional samples provide values used for BSDF sampling, direct lighting calculation, Russian roulette tests, and so forth. Given a particular light transport algorithm (such as path tracing), we can define a mapping between the samples X_i and the particular sample values used for rendering. For example, the first five X_i values might give the camera sample, the next three might be used for sampling the BSDF to find the reflected direction at the first intersection, and so forth.

With the `SamplerRenderer` and the various `Integrators` in `pbrt`, a `Sampler` generates most of the samples X_i—the camera sample in a `CameraSample` structure, and whichever of the additional samples X_i that the surface and volume integrators have requested. Then, the integrators themselves may generate some of their sample values themselves. (For example, the `PathIntegrator` uses samples from the sampler for the first few bounces, but then just generates uniform random numbers with a `RNG` for subsequent bounces.) In the `MetropolisRenderer` implemented in this section, no `Sampler` at all is used and the renderer is responsible for both generating all of the samples and evaluating their contributions.

Given the formalism of an explicit sample vector and the radiance function, we can apply Metropolis to generating the vectors of samples $(X_1, X_2, \ldots)$ for rendering an image. Recall that Metropolis sampling generates samples according to the distribution of the function being sampled. This is an intuitively desirable property: more sampling effort is naturally placed in parts of the sampling space where more light transport is occurring. The state space Ω for this problem is thus the sample vectors $(X_1, X_2, \ldots) \in [0, 1]^\infty$, where L's value is zero for samples outside of the image's extent.[9] For brevity in the equations below, we will write the state vector as

$$\mathbf{X} = (X_1, X_2, \ldots).$$

Metropolis sampling generates samples from the distribution of a given scalar function, so there are two issues that must be addressed in order to use Metropolis for rendering. First, we need to use these samples to compute integrals of the radiance function, and, second, we need to handle the fact that L is a spectrally valued function but Metropolis needs a scalar function.

We can apply the ideas from Section 13.4.4 to use Metropolis samples to compute integrals. First, we define the *image contribution function* such that for an image with j pixels, each pixel I_j has a value that is the integral of the product of the pixel's image reconstruction filter h_j and the radiance L that contributes to the image:

$$I_j = \int_\Omega h_j(\mathbf{X})\, L(\mathbf{X})\, d\Omega.$$

`CameraSample` 342
`Integrator` 740
`MetropolisRenderer` 852
`PathIntegrator` 766
`RNG` 1003
`Sampler` 340
`SamplerRenderer` 25

8 Although L is an infinite-dimensional function, only a finite number of samples of X_i will ever be needed due to path termination with Russian roulette.

9 In the implementation to follow, the film samples are scaled to be in the range of pixel coordinates, and time samples are scaled to be in the camera shutter open time range. However, for the mathematical definition of MLT here, we will operate in terms of all samples being in $[0, 1]$.

Generally, the filter function h_j only depends on whichever two samples of $\mathbf{X}$ that give the image sample position. Further, the value of h_j for any particular pixel is usually zero for the vast majority of samples $\mathbf{X}$ due to the filter's finite extent.

If N samples $\mathbf{X}_i$ are generated from some distribution, $\mathbf{X}_i \sim p(\mathbf{X})$, then the standard Monte Carlo estimate of I_j is

$$I_j \approx \frac{1}{N} \sum_{i=1}^{N} \frac{h_j(\mathbf{X}_i)\, L(\mathbf{X}_i)}{p(\mathbf{X}_i)}.$$

Because L is a spectrally valued function, Metropolis can't be applied directly to generating samples from its distribution; there is no unambiguous notion of what it means to generate samples according to L's distribution. We can, however, define a *scalar contribution function* $I(\mathbf{X})$ to be the function that defines the distribution of samples that are taken with Metropolis sampling. It is desirable that this function be large when L is large so that the distribution of samples has some relationship to the important regions of L. As such, using the luminance of the radiance value is a good choice for the scalar contribution function. In general, any function that is nonzero when L is nonzero will generate correct results, just possibly not as efficiently as a function that is more directly proportional to L.

Given a suitable scalar contribution function, $I(\mathbf{X})$, Metropolis generates a sequence of samples $\mathbf{X}_i$ from I's distribution, the normalized version of I:

$$p(\mathbf{X}) = \frac{I(\mathbf{X})}{\int_\Omega I(\mathbf{X})\, \mathrm{d}\Omega},$$

and the pixel values can thus be computed as

$$I_j \approx \frac{1}{N} \sum_{i=1}^{N} \frac{h_j(\mathbf{X}_i)\, L(\mathbf{X}_i)}{I(\mathbf{X}_i)} \left(\int_\Omega I(\mathbf{X})\, \mathrm{d}\Omega \right).$$

The integral of I over the entire domain Ω can be computed using a traditional approach like path tracing. If this value is denoted by b, with $b = \int I(\mathbf{X})\, \mathrm{d}\Omega$, then each pixel's value is given by

$$I_j \approx \frac{b}{N} \sum_{i=1}^{N} \frac{h_j(\mathbf{X}_i)\, L(\mathbf{X}_i)}{I(\mathbf{X}_i)}. \tag{15.21}$$

In other words, we can use Metropolis sampling to generate samples $\mathbf{X}_i$ from the distribution of the scalar contribution function I. For each sample, the pixels it contributes to (based on the extent of the pixel filter function h) have the value

$$\frac{b}{N} \frac{h_j(\mathbf{X}_i)\, L(\mathbf{X}_i)}{I(\mathbf{X}_i)}$$

added to them. Thus, brighter pixels have larger values than dimmer pixels due to more samples contributing to them (as long as the ratio $L(\mathbf{X}_i)/I(\mathbf{X}_i)$ is generally of the same magnitude). The `Film::Splat()` method defined in Section 7.8.1 is used to accumulate these contributions.

Film::Splat() 403

The MetropolisRenderer, implemented in the remainder of this section, applies Metropolis sampling and Equation (15.21) to render images, using either standard path tracing or bidirectional path tracing to compute the radiance function L. It is defined in the files renderers/metropolis.h and renderers/metropolis.cpp.

15.7.1 SAMPLE REPRESENTATION

We will first define a representation of the vector of sample values $\mathbf{X}$ used for evaluating the radiance value. While the samples could just be represented as a single vector<float>, the MLT implementation below is easier to understand if the sample vector is refined into sections that are more clearly named.

First, instances of the PathSample structure are used when constructing paths of intersection points on surfaces in the scene (starting either from the camera or from a light). In addition to the sample values needed to sample an outgoing direction with the BSDF, rrSample is available for terminating the path with Russian roulette.

⟨*Metropolis Local Declarations*⟩ ≡
```
struct PathSample {
    BSDFSample bsdfSample;
    float rrSample;
};
```

Similarly, LightingSample has all of the information needed to compute direct illumination at a point with multiple importance sampling. The BSDFSample is used for sampling a direction with the BSDF, and the remaining samples are used for selecting a light source and a point on it. Note that the BSDF sample here is used only for computing direct lighting with multiple importance sampling—it is not used in sampling the sequence of intersection points of a light-carrying path.

⟨*Metropolis Local Declarations*⟩ +≡
```
struct LightingSample {
    BSDFSample bsdfSample;
    float lightNum;
    LightSample lightSample;
};
```

The MLTSample structure puts all of the sample values together; it represents a complete vector of sample values. The CameraSample from Section 7.2.1 is used to compute the initial ray leaving the camera. If path tracing is used to evaluate the radiance function, then the cameraPathSamples are used to generate a path from the camera. At each vertex of the path, the sample values in the corresponding lightingSamples element are used to compute direct lighting at the point.

When bidirectional path tracing is used, lightNumSample is used to select which light source to start the path from and the lightRaySamples are used to generate the initial ray leaving the light. Then, lightPathSamples are used to generate the vertices of the path leaving the light. Figure 15.29 summarizes how the various samples are used in constructing light paths and computing their contributions.

BSDFSample 705
CameraSample 342
LightSample 710
MetropolisRenderer 852

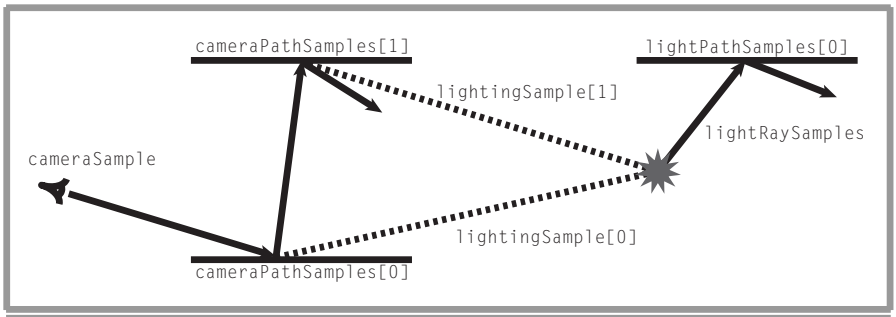


Figure 15.29: Contributions of Members of the `MLTSample` Structure to the Definition of a Path through the Scene. The `cameraSample` defines the camera ray leaving the camera, and the `cameraPathSamples` define how to sample the outgoing directions at the vertices of the camera path. At each camera path vertex, the corresponding `lightingSamples` sample is used for a direct lighting calculation. If bidirectional path tracing is being used, then `lightNumSample` and `lightRaySamples` are used to generate a ray leaving a light source, and `lightPathSamples` are used to sample outgoing directions at light path vertices.

For implementation simplicity, the `MLTSample` allocates storage for samples up to a predetermined maximum path length. To support arbitrary-length light-carrying paths, the lengths of the sample arrays could instead grow on demand and only the samples needed could be generated. See Exercise 15.24 on lazy generation of sample values for MLT for further discussion of this issue.

⟨*Metropolis Local Declarations*⟩ +≡

```
struct MLTSample {
    MLTSample(int maxLength) {
        cameraPathSamples.resize(maxLength);
        lightPathSamples.resize(maxLength);
        lightingSamples.resize(maxLength);
    }
    CameraSample cameraSample;
    float lightNumSample, lightRaySamples[5];
    vector<PathSample> cameraPathSamples, lightPathSamples;
    vector<LightingSample> lightingSamples;
};
```

15.7.2 MUTATIONS

The `MetropolisRenderer` applies two different mutations to the `MLTSample` values. The first (the "large step" mutation) computes new uniform random samples for all of the sample values in **X**. Recall from Section 13.4.2 that it is important that there be greater than zero probability that every possible sample value be proposed; this is taken care of by the large step mutation. In general, the large step mutation helps us widely explore the entire state space without getting stuck on local "islands."

The second mutation (the "small step" mutation) makes a small perturbation to each of the sample values. For difficult lighting configurations, this mutation incrementally

CameraSample 342
LightingSample 838
MetropolisRenderer 852
MLTSample 839
PathSample 838

explores light-carrying paths nearby the current one, so that once a path that makes a significant contribution to the image is found, other paths nearby it can be sampled.

Both of these are symmetric mutations, so their transition probabilities cancel out when the acceptance probability is computed and thus don't need to be computed, as was shown in Equation (13.8).

Implementation of the large step mutation is mostly a matter of calling `RNG::RandomFloat()` to generate new values for all of the samples in the `MLTSample`, with one subtlety related to how new (x, y) image samples are generated, to be explained below.

```
⟨Metropolis Local Declarations⟩ +≡
  static void LargeStep(RNG &rng, MLTSample *sample, int maxDepth,
          float x, float y, float t0, float t1, bool bidirectional) {
      ⟨Do large step mutation of cameraSample 840⟩
      for (int i = 0; i < maxDepth; ++i) {
          ⟨Apply large step to ith camera PathSample 841⟩
          ⟨Apply large step to ith LightingSample⟩
      }
      if (bidirectional) {
          ⟨Apply large step to bidirectional light samples⟩
      }
  }
```

One shortcoming with Metropolis sampling for generating images is that random mutations don't do a good job of ensuring that pixel samples are well stratified over the image plane. In particular, there's no guarantee that all of the pixels will have at least one sample! While the resulting image will still be unbiased in this case, it may be perceptually less pleasing than an image computed with alternative techniques.

To address this issue, `LargeStep()` takes an (x, y) pixel location from the caller; this position is used to set the `imageX` and `imageY` values in the `cameraSample` member variable. It is the caller's responsibility to ensure that in the aggregate, the (x, y) values provided to `LargeStep()` do a good job of sampling the pixels in the image and that they sample the pixels in random order. (For example, if the provided (x, y) values swept across the image in scanline order, taking a single sample in each pixel, the resulting image would be incorrect: that sampling method is not symmetric, while the implementation below expects symmetric transition probabilities.) Ensuring that a minimum set of well-distributed samples are always taken in this manner means that overall image quality improves and a better job is done of (for example) antialiasing geometric edges than would be done otherwise.

```
⟨Do large step mutation of cameraSample⟩ ≡                    840
  sample->cameraSample.imageX = x;
  sample->cameraSample.imageY = y;
  sample->cameraSample.time = Lerp(rng.RandomFloat(), t0, t1);
  sample->cameraSample.lensU = rng.RandomFloat();
  sample->cameraSample.lensV = rng.RandomFloat();
```

CameraSample::imageX 342
CameraSample::imageY 342
CameraSample::lensU 342
CameraSample::lensV 342
CameraSample::time 342
Lerp() 1000
MLTSample 839
MLTSample::cameraSample 839
RNG 1003
RNG::RandomFloat() 1003

Computing new values for the camera path samples is a matter of looping over the remaining elements in the structure and setting their values with `RNG::RandomFloat()`. We thus won't include implementations of the ⟨*Apply large step to i th* `LightingSample`⟩ ⟨*Apply large step to bidirectional light samples*⟩ fragments here since their implementations are essentially all the same.

⟨*Apply large step to i th camera* `PathSample`⟩ ≡ 840
```
PathSample &cps = sample->cameraPathSamples[i];
cps.bsdfSample.uComponent = rng.RandomFloat();
cps.bsdfSample.uDir[0] = rng.RandomFloat();
cps.bsdfSample.uDir[1] = rng.RandomFloat();
cps.rrSample = rng.RandomFloat();
```

The `SmallStep()` function applies a small mutation to the sample value drawn from an exponential distribution as defined by Equation (13.9): $X_i' = X_i \pm b\, e^{-\log(b/a)\,\xi}$, giving a sample in the range $[a, b]$. The advantage of sampling with an exponential distribution like this is that it naturally tries a variety of mutation sizes. It preferentially makes small mutations, close to the minimum magnitudes specified, which help locally explore the path space in small areas of high contribution where large mutations would tend to be rejected. On the other hand, because it also can make larger mutations, it also avoids spending too much time in a small part of the path space in cases where larger mutations have a good likelihood of acceptance.

The `mutate()` utility function mutates the given value using this approach. One important detail is that if the mutated value is outside the range `[min, max]` it wraps around to the other end of the domain: doing so ensures that the transition probabilities for all pairs of sample values are symmetric.

⟨*Metropolis Local Declarations*⟩ +≡
```
static inline void mutate(RNG &rng, float *v, float min = 0.f,
                          float max = 1.f) {
    if (min == max) { *v = min; return; }
    float a = 1.f / 1024.f, b = 1.f / 64.f;
    static const float logRatio = -logf(b/a);
    float delta = (max - min) * b * expf(logRatio * rng.RandomFloat());
    if (rng.RandomFloat() < 0.5f) {
        *v += delta;
        if (*v > max) *v = min + (*v - max);
    }
    else {
        *v -= delta;
        if (*v < min) *v = max - (min - *v);
    }
}
```

BSDFSample::uComponent 705
BSDFSample::uDir 705
MLTSample::cameraPathSamples 839
PathSample 838
PathSample::bsdfSample 838
PathSample::rrSample 838
RNG 1003
RNG::RandomFloat() 1003

The `SmallStep()` function uses `mutate()` to mutate each of the sample values. Here only the mutation of the camera sample is shown; the remainder of the samples are handled in the ⟨*Apply small step mutation to camera, lighting, and light samples*⟩ fragment, not included here.

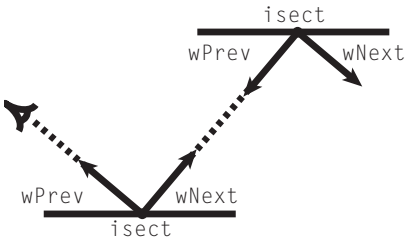


Figure 15.30: Values Stored in the PathVertex at Each Intersection in a Path.

⟨*Metropolis Local Declarations*⟩ +≡

```
static void SmallStep(RNG &rng, MLTSample *sample, int maxDepth,
        int x0, int x1, int y0, int y1, float t0, float t1,
        bool bidirectional) {
    mutate(rng, &sample->cameraSample.imageX, x0, x1);
    mutate(rng, &sample->cameraSample.imageY, y0, y1);
    mutate(rng, &sample->cameraSample.time, t0, t1);
    mutate(rng, &sample->cameraSample.lensU);
    mutate(rng, &sample->cameraSample.lensV);
    ⟨Apply small step mutation to camera, lighting, and light samples⟩
}
```

15.7.3 GENERATING PATHS

Given the few functions that generate and mutate sample vectors above, it's also useful to have a routine that transforms samples into paths through the scene. The `GeneratePath()` function below takes an initial ray (sampled from the camera or a light) and a vector of `PathSample`s that it uses to sample a path through the scene. At each vertex of the path, an instance of the `PathVertex` structure is initialized to store the information about the surface intersection at the corresponding vertex. Figure 15.30 illustrates some of the values stored.

The geometric intersection information for the vertex is stored in the `Intersection`, and normalized vectors to the previous and next vertex in the path are stored in `wPrev` and `wNext`.[10] The `BSDF` at the vertex is stored in `bsdf`. `specularBounce` indicates whether the BSDF component that determined the outgoing direction at this vertex was a delta distribution, and `nSpecularComponents` stores how many specular components the `BSDF` has; the information from these will be needed when computing path contributions with bidirectional path tracing. Finally, the `alpha` member stores the path throughput up to the current vertex (recall the definition of path throughput from Equation (15.7)), scaled by either the emitted light if the path is a light path or the importance from

BSDF 478
CameraSample::imageX 342
CameraSample::imageY 342
CameraSample::lensU 342
CameraSample::lensV 342
CameraSample::time 342
MLTSample 839
MLTSample::cameraSample 839
mutate() 841
PathSample 838
PathVertex 843
RNG 1003

10 Note that this representation is slightly redundant, in that, when present, the previous vertex's `wNext` vector is equal to the negated `wPrev` vector of a vertex. However, this small redundancy is worth the more straightforward code it makes possible.

the camera (here, the ray weight returned by `Camera::GenerateRayDifferential()`) for camera paths.

⟨*Metropolis Local Declarations*⟩ +≡
```
struct PathVertex {
    Intersection isect;
    Vector wPrev, wNext;
    BSDF *bsdf;
    bool specularBounce;
    int nSpecularComponents;
    Spectrum alpha;
};
```

The `GeneratePath()` method takes a ray and and its initial contribution, either importance W_e from the camera or radiance L_e from the light. It expects that the `PathVertex` pointer passed to it will point to an array of at least `samples.size()` `PathVertex`es; it initializes the vertices and returns the length of the actual path generated. (This length may be less than `samples.size()` if the path is terminated with Russian roulette or if a ray leaves the scene.) If the path does terminate because one of the rays traced along the way didn't intersect any scene geometry, this ray is returned in `*escapedRay` (if non-`NULL`) and the current `alpha` value is returned in `*escapedAlpha`.

⟨*Metropolis Method Definitions*⟩ ≡
```
static uint32_t GeneratePath(const RayDifferential &r,
        const Spectrum &a, const Scene *scene, MemoryArena &arena,
        const vector<PathSample> &samples, PathVertex *path,
        RayDifferential *escapedRay, Spectrum *escapedAlpha) {
    RayDifferential ray = r;
    Spectrum alpha = a;
    if (escapedAlpha) *escapedAlpha = 0.f;
    uint32_t length = 0;
    for (; length < samples.size(); ++length) {
        ⟨Try to generate next vertex of ray path 843⟩
    }
    return length;
}
```

BSDF 478
Intersection 186
MemoryArena 1015
PathSample 838
PathVertex 843
PathVertex::isect 843
RayDifferential 69
Scene 22
Scene::Intersect() 23
Spectrum 263
Vector 57

Each time through the loop, `ray` holds either the provided ray `r` (for the first bounce), or the ray leaving the previous vertex of the path. If this ray doesn't intersect any scene geometry, processing here is done. Otherwise, the `path` array is updated with information about the intersection found before a new ray leaving that intersection is sampled.

⟨*Try to generate next vertex of ray path*⟩ ≡ 843
```
PathVertex &v = path[length];
if (!scene->Intersect(ray, &v.isect)) {
    ⟨Handle ray that leaves the scene during path generation 844⟩
}
⟨Record information for current path vertex 844⟩
⟨Sample direction for outgoing Metropolis path direction 844⟩
⟨Terminate path with RR or prepare for finding next vertex 844⟩
```

If the path ends because the current ray doesn't intersect any objects in the scene, the ray and its throughput α are returned to the caller if the respective pointers are non-NULL.

⟨*Handle ray that leaves the scene during path generation*⟩ ≡ 843
```
if (escapedAlpha) *escapedAlpha = alpha;
if (escapedRay)   *escapedRay = ray;
break;
```

Otherwise, an intersection has been found and the PathVertex::isect member has already been initialized by the call to Scene::Intersect(). The current throughput, the BSDF at the intersection, and the previous direction are now stored in the vertex.

⟨*Record information for current path vertex*⟩ ≡ 843
```
v.alpha = alpha;
BSDF *bsdf = v.isect.GetBSDF(ray, arena);
v.bsdf = bsdf;
v.wPrev = -ray.d;
```

The outgoing direction from the vertex is found using the usual BSDF sampling routines; note that the sample values to initialize the BSDFSample are taken from the appropriate PathSample for the current vertex of the path.

⟨*Sample direction for outgoing Metropolis path direction*⟩ ≡ 843
```
float pdf;
BxDFType flags;
Spectrum f = bsdf->Sample_f(-ray.d, &v.wNext, samples[length].bsdfSample,
                            &pdf, BSDF_ALL, &flags);
v.specularBounce = (flags & BSDF_SPECULAR) != 0;
v.nSpecularComponents = bsdf->NumComponents(BxDFType(BSDF_SPECULAR |
                                BSDF_REFLECTION | BSDF_TRANSMISSION));
if (f.IsBlack() || pdf == 0.f)
    return length+1;
```

Finally, pathScale is computed: it gives the incremental factor that updates the path throughput alpha. The magnitude of pathScale is used in a Russian roulette test: the lower it is, the more likely that the path will be terminated. If the path is not terminated, the throughput is updated appropriately, accounting for both pathScale and the adjustment term that compensates for the Russian roulette test. Before the next pass through the loop, the ray to be traced to find the next path vertex is initialized.

⟨*Terminate path with RR or prepare for finding next vertex*⟩ ≡ 843
```
const Point &p = bsdf->dgShading.p;
const Normal &n = bsdf->dgShading.nn;
Spectrum pathScale = f * AbsDot(v.wNext, n) / pdf;
float rrSurviveProb = min(1.f, pathScale.y());
if (samples[length].rrSample > rrSurviveProb)
    return length+1;
alpha *= pathScale / rrSurviveProb;
ray = RayDifferential(p, v.wNext, ray, v.isect.rayEpsilon);
```

AbsDot() 61
BSDF 478
BSDF::dgShading 479
BSDF::Sample_f() 706
BSDFSample 705
BSDF_ALL 428
BSDF_REFLECTION 428
BSDF_SPECULAR 428
BSDF_TRANSMISSION 428
BxDFType 428
DifferentialGeometry::nn 102
DifferentialGeometry::p 102
Intersection::GetBSDF() 484
Intersection::rayEpsilon 186
Normal 65
PathSample 838
PathSample::bsdfSample 838
PathSample::rrSample 838
PathVertex::alpha 843
PathVertex::bsdf 843
PathVertex::isect 843
PathVertex::specularBounce 843
PathVertex::wNext 843
PathVertex::wPrev 843
Point 63
Ray::d 67
RayDifferential 69
Spectrum 263
Spectrum::IsBlack() 265
Spectrum::y() 273

15.7.4 PATH CONTRIBUTIONS

Next, we will define the method that computes the radiance $L(\mathbf{X})$ for a given vector of sample values that represent light-carrying paths of a sequence of points on surfaces in the scene. This method first uses the `GeneratePath()` method to convert the `PathSamples` for the camera and possibly a light source into one or two paths over surfaces in the scene before computing the radiance they transport. The `cameraPath` and `lightPath` parameters should be allocated by the caller to each have length at least equal to `MetropolisRenderer::maxDepth` (though `lightPath` can be `NULL` if bidirectional path tracing isn't being used).

⟨*Metropolis Method Definitions*⟩ +≡

```
Spectrum MetropolisRenderer::PathL(const MLTSample &sample,
        const Scene *scene, MemoryArena &arena, const Camera *camera,
        const Distribution1D *lightDistribution,
        PathVertex *cameraPath, PathVertex *lightPath,
        RNG &rng) const {
    ⟨Generate camera path from camera path samples 845⟩
    if (!bidirectional) {
        ⟨Compute radiance along path using path tracing 846⟩
    }
    else {
        ⟨Sample light ray and apply bidirectional path tracing 846⟩
    }
}
```

Regardless of whether regular path tracing or bidirectional path tracing is being used to compute values of the radiance function, it is necessary to compute a path of intersections with scene surfaces starting from the camera. After the initial ray is generated by the camera, its ray differentials are scaled to account for the actual average number of samples being taken in each pixel; if the samples are equidistantly spaced in both dimensions, their average distance will be one over the square root of the total number of samples taken in the pixel.[11]

⟨*Generate camera path from camera path samples*⟩ ≡ 845

```
RayDifferential cameraRay;
float cameraWt = camera->GenerateRayDifferential(sample.cameraSample,
                                                 &cameraRay);
cameraRay.ScaleDifferentials(1.f / sqrtf(nPixelSamples));
RayDifferential escapedRay;
Spectrum escapedAlpha;
uint32_t cameraLength = GeneratePath(cameraRay, cameraWt, scene, arena,
    sample.cameraPathSamples, cameraPath, &escapedRay, &escapedAlpha);
```

Camera 302
Camera::
GenerateRayDifferential() 303
Distribution1D 648
GeneratePath() 843
MemoryArena 1015
MetropolisRenderer::
maxDepth 853
MLTSample 839
MLTSample::
cameraPathSamples 839
MLTSample::cameraSample 839
PathVertex 843
RayDifferential 69
RayDifferential::
ScaleDifferentials() 70
RNG 1003
Scene 22
Spectrum 263

11 In practice, they aren't equidistantly spaced, since they are generated by random mutations that don't ensure stratification (other than the large step mutations). However, for large numbers of samples, this approximation is usually fine: any aliasing due to insufficient filtering from too-small differentials is generally not noticeable due to the large number of samples taken in the area around each pixel.

If regular path tracing is being used, then the `Lpath()` method (to be defined shortly) computes the radiance carried along the sampled path.

⟨*Compute radiance along path using path tracing*⟩ ≡ 845, 846
```
return Lpath(scene, cameraPath, cameraLength, arena,
             sample.lightingSamples, rng, sample.cameraSample.time,
             lightDistribution, escapedRay, escapedAlpha);
```

For bidirectional path tracing, a light source and a ray leaving the light are sampled. If this sampling step fails to generate a ray that carries any radiance, then radiance along the previously sampled camera path is still computed using regular path tracing. A number of cases may cause the sampled light to return a zero-radiance ray; for example, if the light uses a texture map to describe its emission profile, some regions of the texture are black, and a uniform sampling density is used. In this case, the camera path may still carry radiance, so it's worthwhile to do the standard direct lighting calculation at each path vertex that the path tracing code performs.

⟨*Sample light ray and apply bidirectional path tracing*⟩ ≡ 845
```
⟨Choose light and sample ray to start light path 846⟩
if (lightWt.IsBlack() || lightRayPdf == 0.f) {
    ⟨Compute radiance along path using path tracing 846⟩
}
else {
    ⟨Compute radiance along paths using bidirectional path tracing 846⟩
}
```

⟨*Choose light and sample ray to start light path*⟩ ≡ 846
```
float lightPdf, lightRayPdf;
uint32_t lightNum = lightDistribution->SampleDiscrete(sample.lightNumSample,
                                                      &lightPdf);
const Light *light = scene->lights[lightNum];
Ray lightRay;
Normal Nl;
LightSample lrs(sample.lightRaySamples[0], sample.lightRaySamples[1],
                sample.lightRaySamples[2]);
Spectrum lightWt = light->Sample_L(scene, lrs, sample.lightRaySamples[3],
    sample.lightRaySamples[4], sample.cameraSample.time, &lightRay,
    &Nl, &lightRayPdf);
```

Given the usual case of a light-carrying ray being successfully sampled from the light source, `GeneratePath()` traces the corresponding rays through the scene to compute a path of scattering vertices, using the `lightPathSamples` values for sampling the BSDF at each intersection. Given both the camera and light paths now, `Lbidir()` computes the radiance carried along them.

⟨*Compute radiance along paths using bidirectional path tracing*⟩ ≡ 846
```
lightWt *= AbsDot(Normalize(Nl), lightRay.d) / (lightPdf * lightRayPdf);
uint32_t lightLength = GeneratePath(RayDifferential(lightRay), lightWt,
    scene, arena, sample.lightPathSamples, lightPath, NULL, NULL);
```

AbsDot() 61
CameraSample::time 342
Distribution1D::
SampleDiscrete() 650
GeneratePath() 843
Light 606
Light::Sample_L() 608
LightSample 710
MetropolisRenderer::
Lbidir() 849
MetropolisRenderer::
Lpath() 847
MLTSample::cameraSample 839
MLTSample::
lightingSamples 839
MLTSample::
lightNumSample 839
MLTSample::
lightPathSamples 839
MLTSample::
lightRaySamples 839
Normal 65
Ray 66
Ray::d 67
RayDifferential 69
Scene::lights 23
Spectrum 263
Spectrum::IsBlack() 265
Vector::Normalize() 63

```
    return Lbidir(scene, cameraPath, cameraLength, lightPath, lightLength,
        arena, sample.lightingSamples, rng, sample.cameraSample.time,
        lightDistribution, escapedRay, escapedAlpha);
```

The `MetropolisRenderer::Lpath()` method is best understood as a refactored implementation of `PathIntegrator::Li()`. Here, the geometric part of constructing the path from the camera by sampling the BSDF at each intersection point has been handled by the `GeneratePath()` method and the path is stored explicitly in an array of `PathVertex` structures. The `Lpath()` method then computes the light carried by the path by sampling the direct illumination at each vertex, scaling it by the path's throughput at the vertex and summing the resulting values. In contrast, the computation to generate the path and the computation to compute direct illumination at each path vertex are comingled in the `PathIntegrator`'s `Li()` method, though essentially the same computation is performed there.

⟨*Metropolis Method Definitions*⟩ +≡

```
Spectrum MetropolisRenderer::Lpath(const Scene *scene,
        const PathVertex *cameraPath, int cameraPathLength,
        MemoryArena &arena, const vector<LightingSample> &samples,
        RNG &rng, float time, const Distribution1D *lightDistribution,
        const RayDifferential &eRay, const Spectrum &eAlpha) const {
    Spectrum L = 0.;
    bool previousSpecular = true, allSpecular = true;
    for (int i = 0; i < cameraPathLength; ++i) {
        ⟨Initialize basic variables for camera path vertex 847⟩
        ⟨Add emitted light from vertex if appropriate 848⟩
        ⟨Compute direct illumination for Metropolis path vertex 848⟩
        L += Ld;
    }
    ⟨Add contribution of escaped ray, if any 849⟩
    return L;
}
```

BSDF::dgShading 479
DifferentialGeometry::nn 102
DifferentialGeometry::p 102
DirectLightingIntegrator 742
Distribution1D 648
LightingSample 838
MemoryArena 1015
MetropolisRenderer 852
Normal 65
PathIntegrator::Li() 767
PathVertex 843
PathVertex::bsdf 843
Point 63
RayDifferential 69
RNG 1003
Scene 22
Spectrum 263

⟨*Initialize basic variables for camera path vertex*⟩ ≡ 847, 849

```
const PathVertex &vc = cameraPath[i];
const Point &pc = vc.bsdf->dgShading.p;
const Normal &nc = vc.bsdf->dgShading.nn;
```

The `MetropolisRenderer` can be configured to compute direct illumination using the standard approach as implemented by the `DirectLightingIntegrator`, using Metropolis sampling only for indirect illumination. For scenes where the direct lighting is handled well with the usual algorithms, doing so allows the `MetropolisRenderer` to concentrate its sampling effort on the more difficult portions of the problem. However, this option makes the implementations of `Lpath()` and `Lbidir()` more complex than they would be otherwise: with the lighting integral possibly partitioned, it is necessary to be careful to not double-count any potential light-carrying paths and to not miss any potential light-carrying paths.

If direct lighting is handled separately, then the `directLighting` member variable is non-`NULL`. The separate direct lighting computation includes light emitted at the first intersection point seen by the camera, direct illumination at that point, and emission and direct illumination at any surfaces intersected by sequences of perfect specular reflection or transmission events starting from the first intersection point. Given all this, we can determine whether emission at the current path vertex should be included in the radiance value returned by `Lpath()` as follows:

- If direct lighting is being handled separately, then we only include emission if scattering at the last vertex was specular reflection or transmission and if all of the path's vertices haven't been specular. If the last vertex was nonspecular, then we must ignore emission from this surface, since the direct lighting calculation performed as part of path tracing at the previous vertex is responsible for accounting for this portion of the light transport. If all of the vertices in the path up to this one were specular, then the emission here will have been included by the direct lighting integrator and so should also be ignored here.
- If Metropolis sampling is also responsible for computing direct lighting, then emission is included only if the previous scattering event was specular: in the nonspecular case, path tracing's direct lighting calculation at the previous vertex accounts for illumination scattered by nonspecular components of the BSDF.

⟨*Add emitted light from vertex if appropriate*⟩ ≡ **847, 850**
```
if (previousSpecular && (directLighting == NULL || !allSpecular))
    L += vc.alpha * vc.isect.Le(vc.wPrev);
```

Following similar logic, `Lpath()` skips performing a direct lighting computation at the vertex in cases where a separate direct lighting computation was performed and where that computation has accounted for the current type of light transport path.

⟨*Compute direct illumination for Metropolis path vertex*⟩ ≡ **847, 850**
```
Spectrum Ld(0.f);
if (directLighting == NULL || !allSpecular) {
    ⟨Choose light and call EstimateDirect() for Metropolis vertex 848⟩
}
previousSpecular = vc.specularBounce;
allSpecular &= previousSpecular;
```

Reflection due to direct lighting at this vertex is computed using the regular direct lighting routines. A single light is sampled according to the lights' relative power, and a sample is taken from it using the sample values in the appropriate `LightingSample`. The `alpha` component of the current `PathVertex` provides the scaling factor that accounts for scattering by the BSDFs at the previous vertices in the path.

⟨*Choose light and call* `EstimateDirect()` *for Metropolis vertex*⟩ ≡ **848**
```
const LightingSample &ls = samples[i];
float lightPdf;
uint32_t lightNum = lightDistribution->SampleDiscrete(ls.lightNum,
                                                      &lightPdf);
const Light *light = scene->lights[lightNum];
```

BSDF_ALL 428
BSDF_SPECULAR 428
BxDFType 428
Distribution1D::
SampleDiscrete() 650
EstimateDirect() 749
Intersection::Le() 625
Intersection::rayEpsilon 186
Light 606
LightingSample 838
LightingSample::
bsdfSample 838
LightingSample::lightNum 838
LightingSample::
lightSample 838
MetropolisRenderer::
directLighting 853
PathVertex 843
PathVertex::alpha 843
PathVertex::isect 843
PathVertex::
specularBounce 843
PathVertex::wPrev 843
Scene::lights 23
Spectrum 263

```
Ld = vc.alpha *
     EstimateDirect(scene, this, arena, light, pc, nc, vc.wPrev,
                    vc.isect.rayEpsilon, time, vc.bsdf, rng,
                    ls.lightSample, ls.bsdfSample,
                    BxDFType(BSDF_ALL & ~BSDF_SPECULAR)) / lightPdf;
```

Finally, if the camera path ended because one of the rays in the path didn't intersect any primitives and left the scene, it may be necessary to add the contribution of emitted light from infinite light sources. As before, this computation is only performed if this illumination wasn't accounted for by a separate direct lighting computation or the direct lighting computation performed at the previous vertex.

⟨*Add contribution of escaped ray, if any*⟩ ≡ 847, 849

```
if (!eAlpha.IsBlack() && previousSpecular &&
    (directLighting == NULL || !allSpecular))
    for (uint32_t i = 0; i < scene->lights.size(); ++i)
       L += eAlpha * scene->lights[i]->Le(eRay);
```

`Lbidir()` uses bidirectional path tracing to compute the radiance carried by a camera path and a light path. Recall the discussion from Section 15.3.5 of how the estimates of the path integral form of the light transport equation can be evaluated with a bidirectional approach: for each prefix of the camera path and the light path, a shadow ray is traced between the corresponding two path vertices to construct a light-carrying path. If the points are mutually visible, the full path's contribution is added to the estimate.

⟨*Metropolis Method Definitions*⟩ +≡

```
Spectrum MetropolisRenderer::Lbidir(const Scene *scene,
        const PathVertex *cameraPath, int cameraPathLength,
        const PathVertex *lightPath, int lightPathLength,
        MemoryArena &arena, const vector<LightingSample> &samples,
        RNG &rng, float time, const Distribution1D *lightDistribution,
        const RayDifferential &eRay, const Spectrum &eAlpha) const {
    Spectrum L = 0.;
    bool previousSpecular = true, allSpecular = true;
    ⟨Compute number of specular vertices for each path length 850⟩
    for (int i = 0; i < cameraPathLength; ++i) {
        ⟨Initialize basic variables for camera path vertex 847⟩
        ⟨Compute reflected light at camera path vertex 850⟩
    }
    ⟨Add contribution of escaped ray, if any 849⟩
    return L;
}
```

Distribution1D 648
Light::Le() 631
LightingSample 838
MemoryArena 1015
MetropolisRenderer::directLighting 853
PathVertex 843
RayDifferential 69
RNG 1003
Scene 22
Scene::lights 23
Spectrum 263
Spectrum::IsBlack() 265

When connecting camera paths and light paths to generate a path of particular length in the following, it will be necessary to compute how many other paths of the same length will be created by other bidirectional connections (for example, by using one fewer camera path vertex and one more light path vertex, and so forth). Path vertices with specular reflection or transmission introduce a complication to this computation: if one of the two respective BSDFs at vertices being connected is specular, then we don't bother

tracing a ray to generate a path, since the value of the BSDF is guaranteed to be zero. However, this means that there is one fewer way of generating a path of the corresponding length than there would have been otherwise—this must be accounted for when adding up all of the different ways of generating paths of this length.

To help with this computation, here we initialize an array, `nSpecularVertices`, such that the ith element records how many of the paths with i vertices that can be generated from this camera and light path have specular BSDFs. Note that the first two entries of this array are always zero; while this could be avoided by allocating a smaller array and changing its indexing below, we prefer a simpler indexing scheme that can be easily understood when used in code below.

⟨*Compute number of specular vertices for each path length*⟩ ≡ 849

```
int nVerts = cameraPathLength + lightPathLength + 2;
int *nSpecularVertices = ALLOCA(int, nVerts);
memset(nSpecularVertices, 0, nVerts * sizeof(int));
for (int i = 0; i < cameraPathLength; ++i)
    for (int j = 0; j < lightPathLength; ++j)
        if (cameraPath[i].specularBounce || lightPath[j].specularBounce)
            ++nSpecularVertices[i+j+2];
```

Emission and direct lighting at each camera vertex are computed in the same manner as in the `Lpath()` method, reusing the same code fragments. There are two important details here. First, the direct lighting contribution at this vertex computed by the ⟨*Compute direct illumination for Metropolis path vertex*⟩ fragment, `Ld`, is scaled by the reciprocal of the number of vertices in the path $(i + 1)$ minus the number of paths of this length that end with specular vertices. This factor accounts for all of the different ways that bidirectional path tracing may generate a light-carrying path of the same length.

To understand this factor, consider the direct illumination calculation at the second vertex, $i = 1$, of the camera path: direct lighting at this vertex corresponds to illumination that has bounced off of two surfaces in the scene between the camera and the light. However, bidirectional path tracing can also construct a two-bounce path by connecting the first vertex in the camera path with the first vertex in the light path with a shadow ray. There are thus two ways of constructing two bounce paths, so these two paths must be reweighted with weights that sum to one so that they don't double-count this mode of light transport; a straightforward choice is to weight each of them by 1/2. However, if either or both of the first vertices in the camera or light path was specular, then the corresponding path would have no contribution, and the appropriate weight would be $1/(2 - 1) = 1$. In general, in the absence of specular reflection, there are n ways for bidirectional path tracing to construct an n-bounce light-carrying path.

⟨*Compute reflected light at camera path vertex*⟩ ≡ 849

```
⟨Add emitted light from vertex if appropriate 848⟩
⟨Compute direct illumination for Metropolis path vertex 848⟩
L += Ld / (i + 1 - nSpecularVertices[i+1]);
if (!vc.specularBounce) {
    ⟨Loop over light path vertices and connect to camera vertex 851⟩
}
```

ALLOCA() 1009

PathVertex::
specularBounce 843

If the current camera path vertex is nonspecular, then the corresponding paths connecting it to each of the light path vertices are created.

⟨*Loop over light path vertices and connect to camera vertex*⟩ ≡ 850
```
for (int j = 0; j < lightPathLength; ++j) {
    const PathVertex &vl = lightPath[j];
    const Point &pl = vl.bsdf->dgShading.p;
    const Normal &nl = vl.bsdf->dgShading.nn;
    if (!vl.specularBounce) {
        ⟨Compute contribution between camera and light vertices 851⟩
    }
}
```

Given a particular path, we must determine how to compute the overall contribution of the path we've constructed. Specifically, we are evaluating a particular term of the path integral form of the light transport equation, Equation (15.5), where the path length n is the sum of the number of camera and light vertices in the current path. If we consider the path contribution Equation (15.6), we have constructed camera and light subpaths with the incremental path construction approach used in Section 15.3.3 for regular path tracing. Given the throughput of these paths up to the current vertices, $T(\bar{\mathrm{p}}_i)$ and $T(\bar{\mathrm{q}}_j)$, respectively, we can find that the contribution of a path of i camera vertices and j light vertices is given by

$$P(\bar{\mathrm{p}}_{i,j}) = W_{\mathrm{e}}\, T(\bar{\mathrm{p}}_i) \left[f(\mathrm{p}_{i-1} \to \mathrm{p}_i \to \mathrm{q}_j) G(\mathrm{p}_i \leftrightarrow \mathrm{q}_j) f(\mathrm{p}_i \to \mathrm{q}_j \to \mathrm{q}_{j-1}) \right] T(\bar{\mathrm{q}}_j)\, L_{\mathrm{e}}.$$

The values of two throughput-based terms, $W_{\mathrm{e}} T(\bar{\mathrm{p}}_i)$ for the camera path and $T(\bar{\mathrm{q}}_j) L_{\mathrm{e}}$ for the light path, are already available in `cameraPath[i].alpha` and `lightPath[j].alpha`, respectively, so we only need to compute the term in brackets to find the path's overall contribution. The various values needed to do so are available in the two `PathVertex` structures. One additional detail related to specular reflection is that when evaluating a BSDF that also has specular components an additional scaling factor based on the number of specular components is used to scale the nonspecular BSDF value; this accounts for the PDF for sampling this component of the complete BSDF.

⟨*Compute contribution between camera and light vertices*⟩ ≡ 851
```
Vector w = Normalize(pl - pc);
Spectrum fc = vc.bsdf->f(vc.wPrev, w) * (1 + vc.nSpecularComponents);
Spectrum fl = vl.bsdf->f(-w, vl.wPrev) * (1 + vl.nSpecularComponents);
if (fc.IsBlack() || fl.IsBlack()) continue;
Ray r(pc, pl - pc, 1e-3f, .999f, time);
if (!scene->IntersectP(r)) {
    ⟨Compute weight for bidirectional path, pathWt 852⟩
    float G = AbsDot(nc, w) * AbsDot(nl, w) / DistanceSquared(pl, pc);
    L += (vc.alpha * fc * G * fl * vl.alpha) * pathWt;
}
```

As described above, bidirectional path tracing will construct n separate instances of paths with n vertices. In the aggregate, all of the paths of a particular length must be weighted so that collectively the sum of their weights is one so they make the correct contribution.

AbsDot() 61
BSDF::dgShading 479
BSDF::f() 481
DifferentialGeometry::nn 102
DifferentialGeometry::p 102
DistanceSquared() 65
Normal 65
PathVertex 843
PathVertex::alpha 843
PathVertex::bsdf 843
PathVertex::nSpecularComponents 843
PathVertex::specularBounce 843
PathVertex::wPrev 843
Point 63
Ray 66
Scene::IntersectP() 24
Spectrum 263
Spectrum::IsBlack() 265
Vector 57
Vector::Normalize() 63

In the implementation here, each path is weighted equally. A more effective approach is to weight them using multiple importance sampling, but this improvement is left for Exercise 15.26.

There are two things to note in this computation: first, 2 is added to the sum of i and j to get the number of path vertices. Because arrays are indexed from zero, doing so gives the actual number of vertices in the path. Second, the number of specular vertices in paths of the same length as this one is subtracted from the weight to account for the fact that we don't connect the two paths when we encounter specular vertices, so we are actually making fewer than n connections for paths with n vertices in that case.

⟨*Compute weight for bidirectional path,* `pathWt`⟩ ≡ 851
```
float pathWt = 1.f / (i + j + 2 - nSpecularVertices[i+j+2]);
```

15.7.5 `MetropolisRenderer` IMPLEMENTATION

Given all of this infrastructure—an explicit representation of an n-dimensional sample $\mathbf{X}$, functions to apply mutations to it, and the ability to evaluate the radiance and scalar contribution functions for a given sample value—we can move forward to the heart of the implementation of the `MetropolisRenderer`.

⟨*Metropolis Declarations*⟩ ≡
```
class MetropolisRenderer : public Renderer {
public:
    ⟨MetropolisRenderer Public Methods⟩
private:
    ⟨MetropolisRenderer Private Methods⟩
    ⟨MetropolisRenderer Private Data 853⟩
};
```

The `MetropolisRenderer` constructor, not shown here, initializes the following member variables from parameters provided to it. Most of the member variables are self-explanitory: `camera` holds the `Camera` specified in the scene description file, `bidirectional` is `true` if bidirectional path tracing is to be used to evaluate sample contributions, and if the direct lighting calculation is being done separately from Metropolis sampling then `directLighting` is non-`NULL` and `nDirectPixelSamples` holds the number of samples to be taken per pixel for direct lighting.

The maximum length of a camera path (and light path, if bidirectional path tracing is used) is given by `maxDepth`, and the average number of samples per pixel to take is given by `nPixelSamples`. (In other words, the total number of Metropolis samples taken will be the product of the number of pixels and `nPixelSamples`, but pixels will receive an actual number of samples related to their brightness, due to Metropolis's property of taking more samples in regions where the function's value is high.)

The number of large step mutations for each pixel is given by `largeStepsPerPixel`. Thus, the overall probability of doing a large step mutation rather than a small step mutation is `largeStepsPerPixel / nPixelSamples`.

Camera 302
Renderer 24

Finally, nBootstrap sets the number of "bootstrapping" samples, where the integral of the scalar contribution function is computed, and maxConsecutiveRejects limits the number of successive rejections of proposed mutations; if a large number of mutations are rejected in a row, we override the sample acceptance computation and force acceptance. A long series of rejected mutations should only happen very rarely, but, when it does, image results are improved if we force the sampler to move on, visiting other pixels in the image. This parameter does introduce a small amount of bias in the final result; it should be set to a large value if unbiased rendering is absolutely required.

⟨*MetropolisRenderer Private Data*⟩ ≡ 852
```
Camera *camera;
bool bidirectional;
uint32_t nDirectPixelSamples, nPixelSamples, maxDepth;
uint32_t largeStepsPerPixel, nBootstrap, maxConsecutiveRejects;
DirectLightingIntegrator *directLighting;
```

The Render() method has four main responsibilities. First, if direct lighting is being calculated with the DirectLightingIntegrator, it launches tasks to do this computation. Next, it computes the integral of the scalar contribution function, b. (Recall the need for this value from Equation (15.21).) In order to start Metropolis sampling, a suitable initial sample must be found; this task is handled by the ⟨*Select initial sample from bootstrap samples*⟩ fragment. Finally, this method launches tasks to perform Metropolis sampling before writing out the final rendered image.

There are a few trade-offs to consider in deciding whether to use conventional techniques for the direct lighting lighting calculation. One advantage that the approach implemented in the DirectLightingIntegrator has is that the samples over the light sources can easily be well distributed with low-discrepancy point sets; doing so can substantially improve efficiency. MLT's property of generating samples that are distributed according to the PDF of the function being sampled can also be a disadvantage: if direct lighting makes a substantial contribution to some or all of the image's overall brightness, then MLT will focus the majority of its sampling effort on resolving direct lighting, rather than the potentially more difficult indirect lighting paths. More generally, if there are portions of the radiance function that both make a relatively large contribution and can be handled efficiently with conventional techniques, overall efficiency can be improved if we partition the radiance function so that we can handle the easy parts with conventional approaches and leave the difficult bits for MLT to focus its effort on.

Figure 15.31 compares rendering the San Miguel scene with MLT with direct lighting included and not included, for equal amounts of computation. For this scene, direct lighting is well handled by conventional techniques, so it's more efficient to not include it in the Metropolis sampling phase of rendering.

However, always computing direct lighting separately is not always the right approach. If direct lighting can't be sampled efficiently by standard techniques, then MLT may be a better choice for it. For example, if the majority of the light sources are completely occluded, or if the scene is lit by an infinite area light source with a very bright region

Camera 302
DirectLightingIntegrator 742

(a)

(b)

Figure 15.31: Comparison of Rendering the San Miguel Scene with Direct Lighting Included in Metropolis Sampling and Not Included. (a) Rendered with MLT with 1024 samples per pixel, with direct lighting included in the Metropolis sampling. (b) Rendered with just 900 Metropolis samples per pixel and 128 direct lighting samples per pixel (approximately equal computation time). For this scene, the direct lighting component is well handled by conventional techniques, so it's more efficient to partition the integrand and handle direct lighting separately, allowing the Metropolis sampling phase to concentrate its sampling work on the indirect illumination and not spend its effort on the relatively easy direct lighting. Note that the shadows under the tables are less noisy when direct lighting is done separately, for example. *(Model courtesy of Guillermo M. Leal Llaguno.)*

that happens to be fully occluded, conventional direct lighting methods will take many samples with zero contribution and Metropolis may be a more efficient choice.

Thus, we leave the decision of whether to separately compute direct lighting up to the user. The fragment ⟨*Compute direct lighting before Metropolis light transport*⟩, not included in the text here, launches tasks to compute direct lighting separately if requested. Its structure is quite similar to that of the main `SamplerRenderer` rendering loop in Section 1.3.4.

⟨*Metropolis Method Definitions*⟩ +≡
```
void MetropolisRenderer::Render(const Scene *scene) {
    if (scene->lights.size() > 0) {
        int x0, x1, y0, y1;
        camera->film->GetPixelExtent(&x0, &x1, &y0, &y1);
        float t0 = camera->shutterOpen, t1 = camera->shutterClose;
        Distribution1D *lightDistribution = ComputeLightSamplingCDF(scene);

        if (directLighting != NULL) {
            ⟨Compute direct lighting before Metropolis light transport⟩
        }
        ⟨Take initial set of samples to compute b 855⟩
        ⟨Select initial sample from bootstrap samples 857⟩
        ⟨Launch tasks to generate Metropolis samples⟩
    }
    camera->film->WriteImage();
}
```

Camera::film 302
Camera::shutterClose 302
Camera::shutterOpen 302
ComputeLightSamplingCDF() 709
Distribution1D 648
Film::GetPixelExtent() 404
Film::WriteImage() 404
MemoryArena 1015
MetropolisRenderer::camera 853
MetropolisRenderer::directLighting 853
MetropolisRenderer::maxDepth 853
MetropolisRenderer::nBootstrap 853
MLTSample 839
PathVertex 843
RNG 1003
SamplerRenderer 25
Scene 22
Scene::lights 23

In order to compute the integral of the scalar contribution function $b = \int I(\mathrm{X})\mathrm{d}\Omega$, we can use the sample generation and evaluation routines implemented previously: the `LargeStep()` function is called to generate uniform random samples over the sample space domain, and `PathL()` computes the value of the function for each one. The `sumI` variable accumulates the sum of the scalar contribution functions. This fragment also stores the contribution of each sample in the `bootstrapI` array; these will be used in a second pass through the samples, below, to select the initial sample for rendering.

⟨*Take initial set of samples to compute b*⟩ ≡ 855
```
RNG rng(0);
MemoryArena arena;
vector<float> bootstrapI;
vector<PathVertex> cameraPath(maxDepth, PathVertex());
vector<PathVertex> lightPath(maxDepth, PathVertex());
float sumI = 0.f;
MLTSample sample(maxDepth);
for (uint32_t i = 0; i < nBootstrap; ++i) {
    ⟨Generate random sample and path radiance for MLT bootstrapping 856⟩
    ⟨Compute contribution for random sample for MLT bootstrapping 856⟩
}
float b = sumI / nBootstrap;
```

Recall that `LargeStep()` expects the caller to pass in uniformly sampled x and y pixel coordinates. Here, we don't worry about stratifying those values over the entire image and just compute uniform random values over the pixel range.

⟨*Generate random sample and path radiance for MLT bootstrapping*⟩ ≡ 855
```
float x = Lerp(rng.RandomFloat(), x0, x1);
float y = Lerp(rng.RandomFloat(), y0, y1);
LargeStep(rng, &sample, maxDepth, x, y, t0, t1, bidirectional);
Spectrum L = PathL(sample, scene, arena, camera, lightDistribution,
                   &cameraPath[0], &lightPath[0], rng);
```

After the path's radiance value has been computed, the `I()` function computes its scalar contribution.

⟨*Compute contribution for random sample for MLT bootstrapping*⟩ ≡ 855
```
float I = ::I(L);
sumI += I;
bootstrapI.push_back(I);
arena.FreeAll();
```

As described at the start of the section, the spectral radiance value $L(\mathbf{X})$ must be converted to a value given by the scalar contribution function so that the acceptance probability can be computed for the Metropolis sampling algorithm. There is a fair amount of flexibility in how this function can be defined, as long as it is greater than zero whenever the radiance function is nonzero. For a scene where we wanted to apply extra sampling work to caustics, for example, we could artifically increase its value for any light-carrying path that intersected specular surfaces. Or, if a scene had an extremely bright region that was using too many samples, the function's value could be clamped if it was larger than a fixed limit, so that the Metropolis sampler didn't devote too many samples to the very bright part of the scene. (See the "Further Reading" section for more information on these topics.) Here, we just compute the path's luminance, which is a reasonable baseline approach.

⟨*Metropolis Method Definitions*⟩ +≡
```
inline float I(const Spectrum &L) {
    return L.y();
}
```

Following the approach described in the discussion of start-up bias in Section 13.4.3, here we select an initial sample with probability proportional to its contribution relative to the sum of all of the sample contributions computed above. We can do this without explicitly computing a normalized CDF by selecting a uniform random value between zero and the contribution sum and then looping over all of the path samples from the start until we find the one whose contribution causes the accumulated sum of contributions to pass this value. We don't need to save all of the sample values generated during the first pass; instead we re-seed the random number generator to the same seed as before and then apply the same mutations to the `MLTSample`; the resulting sample will then be the appropriate one.

I() 856
LargeStep() 840
Lerp() 1000
MemoryArena::FreeAll() 1017
MetropolisRenderer::bidirectional 853
MetropolisRenderer::camera 853
MetropolisRenderer::maxDepth 853
MetropolisRenderer::PathL() 845
RNG::RandomFloat() 1003
Spectrum 263
Spectrum::y() 273

⟨*Select initial sample from bootstrap samples*⟩ ≡ 855
```
float contribOffset = rng.RandomFloat() * sumI;
rng.Seed(0);
sumI = 0.f;
MLTSample initialSample(maxDepth);
for (uint32_t i = 0; i < nBootstrap; ++i) {
    float x = Lerp(rng.RandomFloat(), x0, x1);
    float y = Lerp(rng.RandomFloat(), y0, y1);
    LargeStep(rng, &initialSample, maxDepth, x, y, t0, t1,
              bidirectional);
    sumI += bootstrapI[i];
    if (sumI > contribOffset)
        break;
}
```

15.7.6 RENDERING

`MLTTask` provides the `Task` implementation used for the parallel execution of MLT in `pbrt`. Rendering is decomposed so that each task is responsible for performing exactly one large step mutation for each pixel in the image (as well as the appropriate number of small step mutations given the parameter settings). For example, if the overall image resolution is 1024×1024 pixels and the user has set the large step probability to be $1/32$ and the average number of pixel samples to be 512, then $512/32 = 16$ `MLTTasks` will be launched, each one taking a total of $(1024 \times 1024) \times 32$ samples, with $1/32$ of them large steps and $31/32$ of them small steps.

By ensuring that each task performs a number of large step mutations exactly equal to the number of pixels, it is possible to ensure that the (x, y) pixel locations of the proposed large step samples are well distributed across the image plane. Specifically, we make sure that in each task each pixel in the image has exactly one proposed large step mutation. Doing so ensures that all pixels receive at least some samples, which otherwise isn't guaranteed by straightforward application of the basic Metropolis sampling algorithm.

⟨*Metropolis Method Definitions*⟩ +≡
```
void MLTTask::Run() {
    ⟨Declare basic MLTTask variables and prepare for sampling 858⟩
    for (uint64_t s = 0; s < nTaskSamples; ++s) {
        ⟨Compute proposed mutation to current sample 859⟩
        ⟨Compute contribution of proposed sample 860⟩
        ⟨Compute acceptance probability for proposed sample 860⟩
        ⟨Splat current and proposed samples to Film 860⟩
        ⟨Randomly accept proposed path mutation (or not) 861⟩
    }
}
```

A number of values derived from the parameter settings are computed before sampling begins; most of these should be self-explanatory.

LargeStep() 840
Lerp() 1000
MetropolisRenderer::
bidirectional 853
MetropolisRenderer::
maxDepth 853
MetropolisRenderer::
nBootstrap 853
MLTSample 839
RNG::RandomFloat() 1003
RNG::Seed() 1003
Task 1041

⟨*Declare basic* MLTTask *variables and prepare for sampling*⟩ ≡ 857
```
uint32_t nPixels = (x1-x0) * (y1-y0);
uint32_t nPixelSamples = renderer->nPixelSamples;
uint32_t largeStepRate = nPixelSamples / renderer->largeStepsPerPixel;
uint64_t nTaskSamples = uint64_t(nPixels) * uint64_t(largeStepRate);
uint32_t consecutiveRejects = 0;
⟨Declare variables for storing and computing MLT samples 858⟩
⟨Compute L[current] for initial sample 858⟩
⟨Compute randomly permuted table of pixel indices for large steps 859⟩
```

A few variables are also needed for computing MLT sample values and storing the current state. First, memory for PathVertex arrays must be preallocated here so that it can be passed in to calls to the PathL() method below. Storage is also allocated for two MLTSamples, spectral radiance values, and scalar contribution function values. These arrays are indexed with the current and proposed variables; in the implementation below, samples[current] will hold the sample values for the current sample, samples[proposed] will hold the sample values for the proposed mutation, and so forth. When proposed mutations are accepted, the values of current and proposed are swapped. This approach allows us to avoid copying substantial amounts of data when a proposed mutation is accepted, as would be necessary if we instead had distinctly named vector<MLTSample> variables for the proposed and current sample.

⟨*Declare variables for storing and computing MLT samples*⟩ ≡ 858
```
MemoryArena arena;
RNG rng(taskNum);
vector<PathVertex> cameraPath(renderer->maxDepth, PathVertex());
vector<PathVertex> lightPath(renderer->maxDepth, PathVertex());
vector<MLTSample> samples(2, MLTSample(renderer->maxDepth));
Spectrum L[2];
float I[2];
uint32_t current = 0, proposed = 1;
```

The initial sample for all tasks is the one found by the bootstrapping process in Section 15.7.5. All MLTTasks redundantly compute its radiance value and sample contribution, though the cost of this extra work relative to the overall amount of computation the tasks perform is tiny. Because each task uses a unique seed to the pseudo-random number generator, each one will quickly mutate away from the same initial sample to a distinct chain of samples through the state space.

⟨*Compute* L[current] *for initial sample*⟩ ≡ 858
```
samples[current] = initialSample;
L[current] = renderer->PathL(initialSample, scene, arena, camera,
                 lightDistribution, &cameraPath[0], &lightPath[0], rng);
I[current] = ::I(L[current]);
arena.FreeAll();
```

Each pixel in the image should receive a single proposed large step mutation for each task. So that the transition densities between samples for large step mutations is uniform, the order in which pixels are visited by the large step mutations should be randomized.

I() 856
MemoryArena 1015
MemoryArena::FreeAll() 1017
MetropolisRenderer::
  largeStepsPerPixel 853
MetropolisRenderer::
  maxDepth 853
MetropolisRenderer::
  nPixelSamples 853
MetropolisRenderer::
  PathL() 845
MLTSample 839
PathVertex 843
RNG 1003
Spectrum 263

(It should also be different in different tasks.) The bookkeeping for this is handled by assigning each pixel an index and randomly shuffling an array of the pixel indices. The code that calls `LargeStep()` below will in turn convert pixel indices back to (x, y) pixel values.

⟨*Compute randomly permuted table of pixel indices for large steps*⟩ ≡ 858

```
uint32_t pixelNumOffset = 0;
vector<int> largeStepPixelNum;
for (uint32_t i = 0; i < nPixels; ++i) largeStepPixelNum.push_back(i);
Shuffle(&largeStepPixelNum[0], nPixels, 1, rng);
```

The implementation of the Metropolis sampling routine in the following fragments follows the pseudo-code that uses the expected values technique from Section 13.4.1: a mutation is proposed, the value of the function for the mutated sample and the acceptance probability are computed, the contributions of both the new and old samples are recorded, and the proposed mutatation is randomly accepted based on its acceptance probability.

The implementation here regularly alternates between the large step and the small step mutations, taking one large step and then `largeStepRate-1` small steps, another large step, and so forth. Although it could instead randomly choose between the two mutations with corresponding probabilities, with the approach implemented here we ensure that this task will perform exactly one large step mutation for each pixel.

When a large step mutation does occur, the (x, y) image location to pass to the large step is found in a two-step process. First, the current index in the shuffled pixel index array is converted to an integer (x, y) pixel coordinate. Next, an offset in $[0, 1]^2$ is added to the integer pixel coordinate. The same offset is used for *all* of the large steps in this task; the offset values are computed by the code that launches the task and stored in the `MLTTask::dx`, `MLTTask::dy` member variables. To compute these values, the code that launches `MLTTasks` computes a low-discrepancy sampling pattern in $[0, 1]^2$, with one sample for each task. Thus, in the aggregate when all of the tasks have finished, not only will each pixel have received the same number of proposed large step mutations, but also their (x, y) pixel locations will be well distributed within each pixel's area.

⟨*Compute proposed mutation to current sample*⟩ ≡ 857

```
samples[proposed] = samples[current];
bool largeStep = ((s % largeStepRate) == 0);
if (largeStep) {
    int x = x0 + largeStepPixelNum[pixelNumOffset] % (x1 - x0);
    int y = y0 + largeStepPixelNum[pixelNumOffset] / (x1 - x0);
    LargeStep(rng, &samples[proposed], renderer->maxDepth,
              x + dx, y + dy, t0, t1, renderer->bidirectional);
    ++pixelNumOffset;
}
else
    SmallStep(rng, &samples[proposed], renderer->maxDepth,
              x0, x1, y0, y1, t0, t1, renderer->bidirectional);
```

LargeStep() 840
MetropolisRenderer::bidirectional 853
MetropolisRenderer::maxDepth 853
Shuffle() 354
SmallStep() 842

Given the proposed sample, the previously defined `PathL()` and `I()` methods compute the sample's radiance value and scalar contribution.

⟨*Compute contribution of proposed sample*⟩ ≡ 857
```
L[proposed] = renderer->PathL(samples[proposed], scene, arena, camera,
                 lightDistribution, &cameraPath[0], &lightPath[0], rng);
I[proposed] = ::I(L[proposed]);
arena.FreeAll();
```

Since all of the mutations here are symmetric, the transition probabilities cancel when computing the acceptance probability and so the acceptance probability for the proposed sample is found using Equation (13.8).

⟨*Compute acceptance probability for proposed sample*⟩ ≡ 857
```
float a = min(1.f, I[proposed] / I[current]);
```

The contributions for the proposed and current samples are computed using Equation (15.21) and the two samples are splatted into the image using the `Film::Splat()` method. Before being provided to the film, they are scaled with weights based on the expected values optimization introduced in Section 13.4.1.[12] Note that `Film::Splat()` is responsible for incorporating the contribution of filter function $h_j(\mathbf{X})$ before storing the sample's contribution in the pixels that it contributes to.

⟨*Splat current and proposed samples to* `Film`⟩ ≡ 857
```
if (I[current] > 0.f) {
    Spectrum contrib =  (b / nPixelSamples) * L[current] / I[current];
    camera->film->Splat(samples[current].cameraSample,
                        (1.f - a) * contrib);
}
if (I[proposed] > 0.f) {
    Spectrum contrib =  (b / nPixelSamples) * L[proposed] / I[proposed];
    camera->film->Splat(samples[proposed].cameraSample,
                        a * contrib);
}
```

Finally, the proposed mutation is either accepted or rejected, based on the computed acceptance probability `a` (and the fixed limit of successive rejections that are allowed). If the mutation is accepted, then the values of the `current` and `proposed` indices are exchanged; because these variables can have only have the values zero or one, then taking the exclusive-OR of each of them with one does this efficiently.

12 Kelemen et al. (2002) suggested a slightly more efficient implementation of this logic where only the sample that is rejected is splatted to the film, and the contribution of the accepted sample is accumulated in a local variable here until it is eventually rejected. This approach cuts in half the number of `Film::Splat()` calls while still computing the same result.

Camera::film 302
Film::Splat() 403
I() 856
MemoryArena::FreeAll() 1017
MetropolisRenderer::PathL() 845
MLTSample::cameraSample 839
Spectrum 263

⟨*Randomly accept proposed path mutation (or not)*⟩ ≡ 857

```
if (consecutiveRejects >= renderer->maxConsecutiveRejects ||
    rng.RandomFloat() < a) {
    current ^= 1;
    proposed ^= 1;
    consecutiveRejects = 0;
}
else
    ++consecutiveRejects;
```

FURTHER READING

The first application of Monte Carlo to global illumination for creating synthetic images that we are aware of was described in Tregenza's paper on lighting design (Tregenza 1983). Cook's distribution ray-tracing algorithm computed glossy reflections, soft shadows from area lights, motion blur, and depth of field with Monte Carlo sampling (Cook, Porter, and Carpenter 1984; Cook 1986), although the general form of the light transport equation wasn't stated until papers by Kajiya (1986) and Immel, Cohen, and Greenberg (1986). Kajiya's paper also introduced the path-tracing algorithm for solving the light transport equation.

Additional important theoretical work on light transport has been done by Arvo (1993, 1995), who has investigated the connection between rendering algorithms in graphics and previous work in *transport theory*, which applies classical physics to particles and their interactions to predict their overall behavior. Our description of the path integral form of the LTE follows the framework in Veach's Ph.D. thesis, which has thorough coverage of different forms of the LTE and its mathematical structure (Veach 1997). Christensen (2003) wrote a paper on the application of adjoint functions and importance to solving the LTE; it gives a comprehensive overview of related topics with many pointers to further information.

Russian roulette was introduced to graphics by Arvo and Kirk (1990). Hall and Greenberg (1983) had previously suggested adaptively terminating ray trees by not tracing rays with less than some minimum contribution. Arvo and Kirk's technique is unbiased, although in some situations, bias and less noise may be the less undesirable artifact.

A number of approaches have been developed to efficiently render scenes with hundreds or thousands of light sources. (For densely occluded environments, many of the lights in the scene may have little or no contribution to the part of the scene visible from the camera.) Early work on this issue was done by Ward (1991a) and Shirley et al. (1996). Wald et al. (2003) suggested rendering an image with path tracing and a very low sampling rate (e.g., one path per pixel), recording information about which of the light sources made some contribution to the image. This information is then used to set probabilities for sampling each light. Donikian et al. (2006) adaptively found PDFs for sampling lights through an iterative process of taking a number of light samples, noting which ones were effective, and reusing this information at nearby pixels. The "lightcuts" approach, described in a few paragraphs, also addresses this problem.

MetropolisRenderer::
maxConsecutiveRejects 853

RNG::RandomFloat() 1003

pbrt's direct lighting routines are based on using multiple importance sampling to combine samples taken from the BSDF and the light sources' sampling distributions; while this works well in many cases, it can be ineffective in cases where the product of these two functions has a significantly different distribution than either one individually. A number of more efficient approaches have been developed to sample directly from the product distribution (Burke et al. 2005; Ghosh et al. 2005; Cline et al. 2006). Clarberg, Rousselle, and collaborators developed techniques based on representing BSDFs and illumination in the wavelet basis and efficiently sampling from their product (Clarberg et al. 2005; Rousselle et al. 2008). Efficiency of the direct lighting calculation can be further improved by sampling from the *triple product* distribution of BSDF, illumination, and visibility; this issue was investigated by Ghosh and Heidrich (2006) and Clarberg and Akenine-Möller (2008b). Finally, Wang and Åkerlund (2009) have developed a technique that incorporates an approximation to the distribution of indirect illumination in the light sampling distribution used in these approaches.

The general idea of tracing light-carrying paths from the light sources was first investigated by Arvo (1986), who stored light in texture maps on surfaces and rendered caustics. Heckbert (1990b) built on this approach to develop a general ray-tracing-based global illumination algorithm, and Pattanaik and Mudur (1995) developed an early particle-tracing technique. Bidirectional path tracing was independently developed by Lafortune and Willems (1994) and Veach and Guibas (1994). Kollig and Keller (2000) developed the generalization of bidirectional path tracing to use quasi-random sample patterns.

The principles behind the virtual light source approach in the `IGIIntegrator` were first introduced by Keller's work on *instant radiosity* (1997). The instant global illumination algorithm was developed by Wald, Benthin, and collaborators (Wald et al. 2002, 2003; Benthin et al. 2003). The approach for compensating for bias from clamping the geometric term G that was implemented in Section 15.4.2 was developed by Kollig and Keller (2004). For scenes with complex lighting, many of the virtual lights may not make any contribution to the image being generated. To address this issue, Segovia and collaborators (2006, 2007) developed techniques based on bidirectional path tracing and Metropolis sampling to more efficiently generate virtual light sources. The virtual light source approach has recently been successfully used for real-time rendering of scenes with global illumination; see, for example, the papers by Dachsbacher and Stamminger (2006) and Nichols and Wyman (2009).

Building on the virtual point lights concept, Walter and collaborators (2005, 2006) developed *lightcuts*, which are based on creating thousands of virtual point lights and then building a hierarchy by progressively clustering nearby ones together. When a point is being shaded, traversal of the light hierarchy is performed by computing bounds on the error that would result from using clustered values to illuminate the point versus continuing down the hierarchy, leading to an approach with both guaranteed error bounds and good efficiency.

Ward and collaborators developed the irradiance caching algorithm, which is described in a series of papers (Ward, Rubinstein, and Clear 1988; Ward 1994b). One useful improvement to the basic form of the algorithm implemented here is an improved interpolation scheme described by Ward and Heckbert (1992); they estimated the change in irradiance at sample points as the point moves laterally over the surface and as the surface

IGIIntegrator 773

normal changes, based on the individual radiance samples used to compute the irradiance value. This technique results in a smoother interpolated irradiance function. Tabellion and Lamorlette (2004) described a number of additional improvements to irradiance caching that made it effective for rendering for movie productions; our implementation in this chapter generally follows their approach.

Křivánek and collaborators generalized irradiance caching to *radiance caching*, where a more complex directional distribution of incident radiance is stored, so that more accurate shading from glossy surfaces is possible (Křivánek et al. 2005a, 2005b). Gautron et al. (2007) showed how to effectively apply irradiance and radiance caching to rendering animations. Jarosz et al. (2008b) developed an improved gradient computation that better accounts for occlusion and the effect of participating media (i.e., the fact that radiance isn't constant along rays in scenes with volumetric scattering), and Herzog et al. (2009) showed how to use ray differentials for more accurate gradients. Gassenbauer et al. (2009) generalized the idea of caching radiance in the spatial domain to also cache values organized by direction, making the caching approach applicable to highly specular BSDFs as well as diffuse and glossy ones. See also the SIGGRAPH course notes on irradiance caching (Křivánek et al. 2007) and Křivánek and Gautron's book (2009) for a wealth of practical information about implementing these algorithms.

Approaches like Arvo's caustic rendering algorithm (Arvo 1986) formed the basis for an improved technique for caustics developed by Collins (1994). Jensen (1995, 1996a) developed the photon mapping algorithm, which introduced the key innovation of storing the light contributions in a general 3D data structure rather than in textures on surfaces. Important improvements to the photon mapping method are described in follow-up papers by Jensen and collaborators and Jensen's book on photon mapping (Jensen 1996b, 1997, 2001; Jensen and Christensen 1998). Density estimation techniques for global illumination were first introduced by Shirley, Walter, and collaborators (Shirley et al. 1995; Walter et al. 1997).

The question of how to find the most effective set of photons for photon mapping has seen significant amounts of research: light-driven particle-tracing algorithms don't work well for all scenes (consider, for example, a complex building model with lights in every room but where the camera sees only a single room). Early work in this area was done by Peter and Pietrek (1998) and Suykens and Willems (2000b), who developed techniques to store fewer photons in parts of the scene that are relatively unimportant. Fan et al. (2005) showed that the application of Veach's particle-tracing theory to photon mapping that was introduced in Section 15.6.1 demonstrates how photon paths starting from the camera can be generated. They were able to use this approach in conjunction with a Metropolis sampling algorithm to generate photon distributions that were more robust for complicated lighting configurations than those obtained from only following paths from light sources.

Given a set of photons, storing them efficiently is also important for a high-performance implementation. Jensen developed a memory-efficient representation for storing photons and for photon kd-tree nodes (Jensen 1996b), and Steinhurst et al. (2005) developed improved memory layout techniques to improve cache performance and reduce off-chip bandwidth when using photon mapping. Christensen and Batali (2004) developed techniques for storing illumination data for out-of-core use with complex scenes, where a

subset of the data is loaded from disk and cached in memory. Wald et al. (2004) demonstrated that a balanced kd-tree is not optimal for efficient photon lookups. They instead showed how to build kd-trees using a cost metric that estimates the traversal cost when searching for the k nearest neighbors in the tree, building on ideas for building efficient acceleration structures for ray tracing; they demonstrated that this approach led to significantly faster lookups.

Issues related to density estimation and how to best use available photons to compute reflected light have also received significant amounts of attention from researchers. Cammarano and Jensen (2002) showed how to extend photon mapping to support animated scenes. Hey and Purgathofer (2002) investigated density estimation techniques that accounted for the scene geometry where the lookup was being performed, and Schregle (2003) developed a method that chose the number of photons to use for density estimation based on a binary search that tried to find a number that made an optimal trade-off between noise and blurring (using too few and too many photons, respectively). Weber et al. (2004) applied bilateral filtering to photon map density estimates, and Havran et al. (2005a) developed techniques for storing the photon rays in space and interpolating them in this form, rather than storing their intersection points on surfaces. They showed that this approach addresses some of the reconstruction errors that occur when interpolating photons stored on surfaces.

Herzog and Seidel (2007) developed a density estimation kernel that accounts for variation in surface normal and occlusion effects; one source of error from conventional filtering approaches is that part of the area being filtered over may actually be occluded from the viewing perspective. Fabianowski and Dingliana (2009) developed a filtering method based on using ray differentials of the photons to adapt the kernel. Finally, rather than focusing on improved kernels, Spencer and Jones (2009b) had the insight that redistributing the photons so that no two of them are too close together makes it possible to use many fewer of them in estimates while still generating high-quality results.

One way to reduce the memory requirements for storing photons is to store few or none of them. To this end, Havran et al. (2005b) developed a final gathering algorithm based on storing final gather intersection points in a kd-tree in the scene and *then* shooting photons from the lights; when a photon intersects a surface, the nearby final gather intersection records are found and the photon's energy can be distributed to the origins of the corresponding final gather rays; the photon doesn't need to be stored at all. Herzog et al. (2007) described an approach based on storing all of the visible points as seen from the camera and *splatting* photon contributions to them, and Hachisuka et al. (2008b) developed the *progressive photon mapping* algorithm, where the points visible from the camera are stored and a series of iterations are executed where a fixed number of photons are traced and distributed to the points. After the points' reflected radiance values have been updated, the photons are discarded; as more iterations are performed, the values converge to the correct result.

Final gathering for finite-element radiosity algorithms was first described in Reichert's thesis (Reichert 1992). One important use of information available in the photon map is optimized final gathering techniques that importance-sample directions for the gathering step using the photons to give information about which directions are likely to

have large contributions (Jensen 1995; Hey and Purgathofer 2002). Christensen (1999) developed the important optimization for the final gathering step of precomputing outgoing radiance at a subset of the photons that was implemented here in Section 15.6.3. Arikan et al. (2005) described an efficient final gathering approach based on handling nearby and far-away components of the incident radiance field separately. More recently, Spencer and Jones (2009a) described how to build a hierarchical kd-tree of photons such that traversal could be stopped at higher levels of the tree and showed that using the footprints of final gather rays, computed using ray differentials, can lead to better results than the usual approach.

Veach and Guibas (1997) first applied the Metropolis sampling algorithm to solving the light transport equation. They demonstrated how this method could be applied to image synthesis and showed that the result was a light transport algorithm that was robust to traditionally difficult lighting configurations (e.g., light shining through a slightly ajar door). The approach used in the `MetropolisRenderer` to ensure that a given number of samples are proposed in each pixel is the same as the lens subpath mutation that they proposed. Kelemen et al. (2002) developed the formulation of Metropolis rendering used by the `MetropolisRenderer` in this chapter; their approach is more easily implemented than Veach and Guibas's. They also suggest a useful alternative approach for weighting the current and proposed samples based on multiple importance sampling. More recently, Fan et al. (2005) developed a method that let the user explicitly provide a number of important paths (for example, through a tricky geometric configuration) that could then be used in Metropolis sample generation. Cline et al. (2005) developed a light transport algorithm based on generating an independent chain of Metropolis samples in each pixel. Hoberock's Ph.D. dissertation discusses a number of alternatives for the scalar contribution function, including those that adapt the sampling density to pay more attention to particular modes of light transport and those that focus on reducing noise in the final image (Hoberock 2008).

Hair is particularly challenging to render; not only is it extremely geometrically complex, but multiple scattering among hair makes a significant contribution to its final appearance. Traditional light transport algorithms often have difficulty handling this case well. See the papers by Moon and Marschner (2006), Moon et al. (2008), and Zinke et al. (2008) for recent work in specialized rendering algorithms for hair.

EXERCISES

❷ 15.1 To further improve efficiency, Russian roulette can be applied to skip tracing many of the shadow rays that make a low contribution to the final image: to do so, tentatively compute the potential contribution of each shadow ray to the final overall radiance value before tracing the ray. If the contribution is below some threshold, apply Russian roulette to possibly skip tracing the ray. Recall that Russian roulette always increases variance; when evaluating the effectiveness of your implementation, you should consider its *efficiency*—how long it takes to render an image at a particular level of quality.

MetropolisRenderer 852

2 15.2 Read Veach's description of efficiency-optimized Russian roulette, which adaptively chooses a threshold for applying Russian roulette (Veach 1997; Section 10.4.1). Implement this algorithm in `pbrt` and evaluate its effectiveness in comparison to manually setting these thresholds.

3 15.3 Implement a technique for generating samples from the product of the light and BSDF distributions; see the papers by Burke et al. (2005), Ghosh et al. (2005), Cline et al. (2006), Clarberg et al. (2005), and Rousselle et al. (2008). Compare the effectiveness of the approach you implement to the direct lighting calculation currently implemented in `pbrt`. Investigate how scene complexity (and, thus, how expensive shadow rays are to trace) affects the Monte Carlo efficiency of the two techniques.

2 15.4 Clarberg and Akenine-Möller (2008b) described an algorithm that performs visibility caching, computing, and interpolating information about light source visibility at points in the scene. Implement this method and use it to improve the direct lighting calculation in `pbrt`. What sorts of scenes is it particularly effective for? Are there scenes for which it doesn't help?

2 15.5 Investigate algorithms for rendering scenes with large numbers of light sources: see, for example, the papers by Ward (1991a), Shirley, Wang, and Zimmerman (1996), Keller and Wald (2000), and Donikian et al. (2006) on this topic. Choose one of these approaches and implement it in `pbrt`. Run experiments with a number of scenes to evaluate the effectiveness of the approach that you implement.

3 15.6 Modify `pbrt` so that the user can flag certain objects in the scene as being important sources of indirect lighting and modify the `PathIntegrator` to sample points on those surfaces according to dA to generate some of the vertices in the paths it generates. Use multiple importance sampling to compute weights for the path samples, incorporating the probability that they would have been sampled both with BSDF sampling and with this area sampling. How much can this approach reduce variance and improve efficiency for scenes with substantial indirect lighting? How much can it hurt if the user flags surfaces that actually make little or no contribution, or if multiple importance sampling isn't used? Investigate generalizations of this approach that learn which objects are important sources of indirect lighting as rendering progresses so that the user doesn't need to supply this information ahead of time.

3 15.7 Light transport algorithms that trace paths from the lights like bidirectional path tracing and particle tracing implicitly assume that the BSDFs in the scene are symmetric—that $f(\mathrm{p}, \omega, \omega') = f(\mathrm{p}, \omega', \omega)$. However, both real-world BSDFs like the one describing specular transmission as well as synthetic BSDFs like the ones that are used when shading normals are present are actually not symmetric. Not accounting for this situation can lead to errors in the results computed by these light transport algorithms (Veach 1997; Chapter 5). Fix `pbrt` to properly handle nonsymmetric BSDFs. Demonstrate that these fixes eliminate errors that are present in the current implementation.

PathIntegrator 766

● 15.8 The implementation of the IGIIntegrator in Section 15.4 only traces paths starting from the light sources in order to place virtual point lights in the scene. This approach can be ineffective in scenes where most paths from the lights do not generate virtual lights that are visible from the points visible to the camera. Implement one of the improved approaches for placing virtual lights introduced by Segovia et al. (2006, 2007). Show the improvement in image quality that results for a scene where the IGIIntegrator doesn't do well.

● 15.9 Implement Walter et al.'s *lightcuts* algorithm in pbrt (Walter et al. 2005, 2006). How do the BSDF interfaces in pbrt need to be generalized to compute the error terms for light cuts? Do other core system interfaces need to be changed? Compare the efficiency of rendering images with global illumination using your implementation to some of the other integrators.

● 15.10 Read Ward and Heckbert's paper on irradiance gradients (Ward and Heckbert 1992) and implement this technique in the IrradianceCacheIntegrator. How much does it improve the quality of the results that it computes?

● 15.11 Improve the IrradianceCacheIntegrator's computation of irradiance values by modifying it to use bidirectional path tracing instead of regular path tracing to compute the individual radiance values. Construct a scene where this approach is much better than path tracing. How much does it improve the results for scenes where the path-tracing approach works well already?

● 15.12 Modify pbrt to implement the irradiance filtering method described by Kontakten et al. (2004). Compare the results from this approach to the current IrradianceCacheIntegrator implementation.

● 15.13 Implement the radiance caching algorithm developed by Křivánek et al. (2005a, 2005b). Compare the results of rendering glossy indirect reflections with your implementation to the results from the current IrradianceCacheIntegrator: how much more accurate are the results compared with your implementation? Also compare your results to a reference solution computed with the PathIntegrator or MetropolisRenderer: how much more efficiently does your implementation produce these effects?

● 15.14 Experiment with the parameters to the PhotonIntegrator until you get a good feel for how they affect results in the final image. At a minimum, experiment with varying the search radius, the number of photons traced, and the number of photons used.

● 15.15 If photon mapping is being used only to render caustics in a scene, generation of the caustic map can be accelerated by flagging which objects in the scene potentially have specular components in their BSDFs and then building a directional table around each light source, recording which directions could potentially result in specular paths (Jensen 1996a). Photons can only be shot in these directions, saving the work of tracing photons in directions that are certain not to lead to caustics. Extend pbrt to support this technique. Note that different solutions will be needed for point light sources, directional light sources, and area

IGIIntegrator 773
IrradianceCacheIntegrator 786
MetropolisRenderer 852
PathIntegrator 766
PhotonIntegrator 802

light sources. How much does this approach end up improving performance for scenes where it's applicable?

❷ 15.16 Another approach to improving the efficiency of photon shooting is to start out by shooting photons from lights in all directions with equal probability, but then to dynamically update the probability of sampling directions based on which directions lead to light paths that have high throughput and which directions are less effective. Photons then must be reweighted based on the probability for shooting a photon in a particular direction. As long as there is always nonzero possibility of sending a photon in any direction, this approach doesn't introduce bias into the shooting algorithm. One advantage of this approach compared to the one in Exercise 15.15 is that the user of the renderer doesn't need to flag which objects may cast caustics ahead of time.

❸ 15.17 Section 15.6.1 introduces theory that leads to algorithms where photon paths can be generated starting from the camera as well as the lights; this approach was applied to photon mapping by Fan et al. (2005), who also used Metropolis sampling to select paths. Modify the `PhotonIntegrator` implementation to also generate paths from the camera, computing appropriate weights for them. Demonstrate the effectiveness of this approach for scenes where paths from the lights are not an efficient approach.

❸ 15.18 The photons around a point carry additional useful information about the illumination there that can be used for efficient rendering. See the ideas introduced by Keller and Wald (2000) and extend the photon mapping integrator to store direct lighting photons that also store which light source each direct lighting photon originated from. When computing direct illumination at a point, use the nearby direct lighting photons to choose light source sampling probabilities based on the estimated contribution of each of the light sources. Evaluate the effectiveness of this approach with a number of scenes.

❷ 15.19 Read the paper by Wald et al. (2004) on photon kd-tree construction and modify the implementation in `pbrt` to use their approach. How much difference in rendering time do you see when using their method with scenes that do photon mapping? (You may want to experiment with scenes with both smoothly varying indirect lighting as well as those with sharp caustics, which will exhibit relatively more variation in photon density.) Use a profiler to measure how much kd-tree construction time for the photon maps is affected by the new method as well as how the time spent performing photon lookups is affected.

❸ 15.20 Modify the `IrradianceCacheIntegrator` or `PhotonIntegrator` to use adaptive sampling for computing irradiance estimates or final gather contributions, respectively. Take some number of samples and use them to decide if the function being integrated seems to be relatively smooth or quickly changing. If it is smooth, stop sampling, but otherwise take a larger number of samples in an effort to compute a better estimate. Experiment with different criteria for deciding when to take more samples. How much of an efficiency improvement is your approach able to achieve compared to always taking a large number of

`IrradianceCacheIntegrator` 786
`PhotonIntegrator` 802

samples? Read Kirk and Arvo's paper that discusses how adaptive sampling can introduce bias (Kirk and Arvo 1991). In your implementation, is it worthwhile to use an unbiased adaptive sampling approach, or is the bias unobjectionable in practice?

● 15.21 Another approach for shooting photons that can *guarantee* constant weights is to use rejection sampling. Show how to use rejection sampling for shooting photons from lights and scattering photons from surfaces when the sampling distribution doesn't exactly match the function being sampled in a way that guarantees constant-weight photons. In what ways do the light source and BSDF sampling methods need to be modified for this approach?

● 15.22 Read Hachisuka et al.'s paper on progressive photon mapping (Hachisuka et al. 2008b) and implement their algorithm in `pbrt`. In order to maintain information about all of the points visible from the camera over the course of the entire rendering computation, you will likely need to implement their approach as a new `Renderer`. Compare the results with this algorithm to those generated by the regular `PhotonIntegrator` and the `MetropolisRenderer` for scenes with difficult lighting. Parallelize your implementation by breaking the various phases of computation into tasks. How well does performance scale with increasing numbers of processors?

● 15.23 The current implementation of the `MetropolisRenderer` always mutates all of the samples, up to the maximum ray depth that was specified. This approach is needlessly inefficient in that when the path length is shorter than the maximum depth (due to termination from Russian roulette or a ray leaving the scene), the extra samples are never used. However, generating samples lazily must be done carefully for correct results; for newly needed samples, it's necessary to replay the missed mutation history since the last large step. Read Kelemen et al.'s paper, which discusses this issue (Kelemen et al. 2002), and implement their lazy mutation strategy. As part of this work, make it possible for paths to have arbitrary lengths rather than a fixed maximum; doing so will require allowing both the `PathVertex` arrays as well as the `vectors` in the `MLTSample` to be able to grow dynamically.

● 15.24 By adding mutation strategies that don't necessarily modify all of the sample values X_i, it can be possible to reuse some or all of the paths generated by the previous samples in the `MetropolisRenderer`. For example, if only the `LightingSample` values are mutated in the `MLTSample`, then the camera and light paths can be reused and the shadow rays connecting their vertices don't need to be retraced. (Indeed, the `Spectrum` representing the entire indirect lighting contribution found from these connections can be reused.) In a similar manner, if only the light path's sample values are mutated, then not only does the camera path not need to be retraced, but the results from the direct lighting calculations at the camera path's vertices can be reused. As a third alternative, if the samples for the first few vertices of the camera or light path are left unchanged, then that portion of the path doesn't need to be retraced.

LightingSample 838
MetropolisRenderer 852
MLTSample 839
PathVertex 843

Modify the `MetropolisRenderer` to add one or more of the above sampling strategies and modify the implementation so that it reuses any partial results from the previous sample that remain valid when your new mutation is used. You may want to add both "small step" and "large step" variants of your new mutation. Compare the mean squared error of images rendered by your modified implementation to the MSE of images rendered by the original implementation, comparing to a reference image rendered with a large number of samples. For the same number of samples, your implementation should be faster but will likely have slightly higher error due to additional correlation between samples. Is the Monte Carlo efficiency of your modified version better than the original implementation?

● 15.25 In his Ph.D. dissertation, Hoberock proposes a number of alternative scalar contribution functions for Metropolis light transport, including ones that focus on particular types of light transport and ones that adapt the sample density during rendering in order to reduce perceptual error (Hoberock 2008). Read Chapter 6 of his thesis and implement either the multistage MLT or the noise-aware MLT algorithm that he describes.

● 15.26 A significant shortcoming of the bidirectional path-tracing algorithm implemented in the `MetropolisRenderer` is that it equally weights all of the various ways of generating a path of given length (recall the implementation in the fragment ⟨*Compute weight for bidirectional path,* `pathWt`⟩). Veach and Guibas demonstrated that applying multiple importance sampling to weighting bidirectional paths can lead to substantially less variance at relatively small incremental cost (Veach and Guibas 1995; Veach 1997). Modify the `MetropolisRenderer` to use MIS in its `Lbidir()` method and measure the improvement in image quality. How much does execution time increase from your change?

● 15.27 Another shortcoming of the bidirectional path-tracing algorithm implemented in the `MetropolisRenderer` is that it doesn't support paths that start at the light source and have zero camera path vertices. These types of paths are particularly important for rendering caustics. Consider, for example, caustics on a diffuse surface due to light passing through a glass sphere from a point light source: it's not possible to sample an outgoing direction from the diffuse surface that passes through the two delta-distribution BSDFs on the sphere and then intersects the point light—the only way to sample this type of path is to sample a path from the light and add contributions from points on the diffuse surface that are intersected by light paths and are visible to the camera.

Modify the `MetropolisRenderer::Lbidir()` method to support zero-vertex camera paths and modify the `ImageFilm` and `Camera` implementations to support adding contributions from arbitrary points in the scene to the image. (Contributions from points that aren't visible from the camera can just be discarded.) Don't forget to update the path weighting computation since your change will add one more way to generate a path of each length. You will also need to include the effect of scattering in the direction back toward the camera

Camera 302
ImageFilm 404
MetropolisRenderer 852
MetropolisRenderer::
Lbidir() 849

at the last vertex of the light path. One important detail is that with this change a single `MLTSample` may contribute to multiple pixels—one from the camera path and potentially one more for each vertex of the light path. Don't forget to add code to account for this in the `MLTTask::Run()` method.

Render images that show off the improvement from this change when rendering caustics. Does this approach help with efficiency for other indirect lighting cases?

MLTSample 839
MLTTask::Run() 857